HARMONY: AN EXECUTION MODEL FOR HETEROGENEOUS SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Gregory Frederick Diamos

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2011

This dissertation is dedicated to my wife, Sudnya Diamos.

ACKNOWLEDGEMENTS

During my tenure as a graduate student at GaTech, I have been fortunate to work with three exceptional people: Nathan Clark, Karsten Schwan, and Sudhakar Yalamanchili. My ideas about runtime kernel scheduling have been guided and refined by my thesis advisor, Sudhakar Yalamanchili. He also deserves thanks for teaching me to understand the high level structure of a problem before diving into the details. Karsten Schwan helped me form a systems perspective on kernel-based programming models. I am grateful to Nathan Clark for introducing me to the existing body of work on dynamic compilation, giving me a starting point for my research on dynamic compilation for heterogeneous processors. I am also thankful to Gabriel Loh for giving me best overview of modern processor architecture that I have heard so far. I am indebted to John Nickolls for accepting me as an intern at NVIDIA and allowing me to contribute to the design of the Fermi processor. This work was greatly influenced by his perspectives on parallel processor architecture.

I thank all of the members of my reading committee: Douglas Blough, Hsien-hsin Lee, Sudhakar Yalamanchili, Bonnie Ferri, and Karsten Schwan.

My fellow students, Andrew Kerr, Haicheng Wu, Jeffery Young, Nawaf Almoosa, Rick Copeland, Vishakha Gupta, and others have provided constructive criticism, comments, and assistance.

Much of the work in this thesis utilizes the Ocelot open-source compiler and emulator to conduct experiments. I am thankful to all of the Ocelot contributors who donated their valuable time to enhance the project.

Financial support for this work was provided by NVIDIA, LogicBlox, and the National Science Foundation. I am grateful to NVIDIA for a Ph.D. fellowship.

Most of all, this dissertation is dedicated to Sudnya Diamos. Without her support and encouragement, this work would not have been possible.

TABLE OF CONTENTS

DEDICATION **2**

ACKNOWLEDGEMENTS **3**

LIST OF TABLES **11**

LIST OF FIGURES **12**

Glossary **17**

SUMMARY **20**

I INTRODUCTION **21**

1.1 Performance Scaling on Heterogeneous Processors 21

1.2 Execution Models 23

1.3 Relating Structure and Complexity 24

1.4 Roots in Out of Order Execution 27

1.4.1 Instruction dependences 27

1.4.2 Dependency-Driven Instruction Scheduling 27

1.4.3 Branch Prediction 28

1.5 Managing Dynamic Overhead 28

1.6 Towards a Generic Execution Model 29

1.7 Contributions 31

1.8 Thesis Organization 32

II AN EXECUTIVE SUMMARY **34**

2.1 A Sample Machine Model 34

2.2 Matrix Multiplication in Harmony 35

2.3 Execution with the Harmony Runtime 37

III ORIGIN OF THE PROBLEM AND CURRENT LANDSCAPE OF RELATED WORK **41**

3.1 The Rise and Fall of General Purpose Computing 41

3.2 The Many-Core and Heterogeneous Revolution 42

3.2.1 Many-Core Processors 42
3.2.2 Heterogeneous Systems 43
3.3 Programming Languages for Parallel and Heterogeneous Systems . . 44
3.3.1 Implicitly and Explicitly Parallel Programming 45
3.3.2 Stream Programming 46
3.3.3 Bulk-Synchronous-Parallel Languages 46
3.3.4 Heterogeneous-Aware Languages 47
3.4 Dynamic and Heterogeneous Compilers 48
3.4.1 Dynamic Compilers 49
3.4.2 Heterogeneous Compilers 50
3.4.3 Optimizations for Heterogeneous Compilers 51
IV THE HARMONY EXECUTION MODEL 53
4.1 The Harmony Machine Model 54
4.1.1 Processing Elements (Processors) 54
4.1.2 Memory Spaces 56
4.1.3 Central Control 59
4.2 Model Abstractions 62
4.2.1 Compute Kernels 62
4.2.2 Control Decisions 63
4.2.3 Variables 63
4.2.4 Discussion 64
4.3 Model Analysis 66
4.3.1 Kernel dependences 67
4.3.2 The Kernel Control Flow Graph 70
4.3.3 Dependency Graph 71
4.3.4 Performance Analysis 71
4.3.5 Reorganizing an Algorithm 75
4.4 Expressing Applications in Harmony 75

4.4.1 High Level Languages 75
4.5 Examples 79
4.5.1 Petrinet Simulation 80
V APPLICATION OF THE MODEL AT MULTIPLE GRANULARITIES 85
5.1 Functional Units in a Superscalar Processor 85
5.1.1 Pipeline Organization 85
5.1.2 Overheads 89
5.2 Many Core System on Chips 91
5.2.1 System Overview 91
5.2.2 Resource Sharing 93
5.3 Directly Connected Accelerator Boards 94
5.3.1 Compositional Systems 94
5.3.2 Accelerator Classes 96
5.4 Heterogeneous Clusters 98
5.4.1 Node Organization 98
5.4.2 Distributed Memory Spaces 100
5.5 Summary 100
VI MODEL OPTIMIZATIONS 101
6.1 Performance Models 101
6.1.1 Performance Metrics 102
6.1.2 The Kernel Mapping Problem 103
6.1.3 System Characteristics 103
6.1.4 Static Kernel Characteristics 104
6.1.5 Dynamic Kernel Characteristics 105
6.1.6 Predictive Models 106
6.2 Static Optimizations 107
6.2.1 Loop Transformations 108
6.2.2 Variable Allocation 112

6.2.3 Common-Subexpression Elimination/Re-materialization . . . 113
6.2.4 Variable Coalescing . . . 114
6.2.5 Kernel Fusion/Fission . . . 116
6.2.6 Kernel Dependency Graph Partitioning . . . 119
6.2.7 Static Kernel Scheduling . . . 121
6.2.8 Static Predictions . . . 121
6.3 Dynamic Optimizations . . . 121
6.3.1 Speculation . . . 122
6.3.2 On-line Machine Modeling . . . 125
6.3.3 Variable Renaming . . . 126
6.3.4 Kernel and DMA Scheduling . . . 128
6.4 Fault Tolerance . . . 131
6.4.1 Kernels As Transactions . . . 132
6.4.2 Memory Versioning . . . 132
6.4.3 Heterogeneity and Diversity . . . 133
VII THE KERNEL EXECUTION MODEL . . . 134
7.1 Structured Parallelism . . . 135
7.1.1 Explicitly Parallel Models . . . 135
7.1.2 Bulk-Synchronous Parallel Kernels . . . 136
7.1.3 A Bulk-Synchronous Execution Model . . . 136
7.1.4 The PTX Model . . . 138
7.1.5 Translating the PTX Model . . . 140
7.1.6 Thread Fusion . . . 141
7.2 Thread Fusion . . . 143
7.2.1 PTX Thread Serialization . . . 144
7.3 Subkernel Formation . . . 145
7.4 The Ocelot Dynamic Compiler . . . 147
7.4.1 Dynamic Optimization . . . 147

7.5 Alternative Execution Models 148

VIII CASE STUDIES 149

8.1 The Limits of Kernel Level Parallelism 150

8.1.1 Kernel Level Parallelism in Harmony Applications 150

8.2 The Design of The Harmony Runtime 154

8.2.1 The Runtime Components 154

8.2.2 Performance Evaluation 159

8.3 Ocelot: A Prototype Dynamic Kernel Compiler 164

8.3.1 Microbenchmarks 164

8.3.2 Runtime Overheads 170

8.3.3 Full Application Scaling 173

8.4 Subkernel Formation 175

8.4.1 Abstract 175

8.4.2 Introduction 175

8.4.3 Bulk-Synchronous Models 177

8.4.4 The Subkernel 177

8.4.5 Performance Evaluation 180

8.4.6 Concluding Remarks 182

8.5 On-line Kernel Performance Modeling 183

8.5.1 Instrumentation With Ocelot 184

8.5.2 Metrics and Statistics 185

8.5.3 Benchmarks 188

8.5.4 Results 189

8.5.5 Concluding Remarks 208

8.6 A High Level Language Frontend for Harmony 209

8.6.1 Datalog$_{LB}$ 209

8.6.2 Relational Algebra Operators 211

8.6.3 Algorithm Skeletons 212

8.6.4 CUDA Kernel Skeleton Specialization 217
8.6.5 The RedFox Compiler 218
8.6.6 Microbenchmarks 221
8.6.7 Simple Application Performance 222
8.6.8 Insights and Discussion 223
8.6.9 Summary 223
IX MODEL EXTENSIONS AND FUTURE WORK 224
9.1 The Harmony Execution Model 224
9.1.1 Model Abstractions 225
9.1.2 Applying Harmony At Different Scales 225
9.1.3 Optimizing the Model 226
9.2 Important Directions for Future Work 226
9.2.1 A Model of Locality 226
9.2.2 Kernel Specialization 227
9.2.3 Beyond Subkernel Formation 228
9.3 Conclusion 228
REFERENCES 229

LIST OF TABLES

1 Application Characteristics 150

2 Parallelism in CUDA Applications 151

3 Test System 154

4 Kernel Startup And Teardown Overhead 171

5 Benchmark Applications. 188

6 Machine parameters. 190

7 List of metrics. 190

8 Metrics for each of the Parboil benchmark applications using default input sizes. 191

9 The set of relational algebra operations. 211

10 The throughput of each of the algorithm skeletons over random datasets. These operations are assumed to be memory bound, and throughput is counted as the size of the input relations plus that of the output relation divided by the total execution time of the algorithm. 221

11 The microbenchmark test system configuration. All of the benchmarks are run only on the GPU. 221

12 Simple Datalog^{LB} benchmarks. 222

LIST OF FIGURES

1 The data dependency graph for an FFT kernel. 25

2 The data dependency graph for a simple vector addition kernel. . . . 25

3 An example of 3x3 blocked matrix multiply in the Harmony IR. . . . 36

4 The initial state of the fetch buffer for the Harmony 3x3 matrix multilpy example. 37

5 The initial dataflow graph for the Harmony 3x3 matrix multilpy example. The right side of the figure shows the critical path. 38

6 The initial kernel schedule for the Harmony 3x3 matrix multilpy example. The critical path is shown in red. 39

7 GFPLOPs per dollar for different multi-core platforms when running the Nbody application developed by Arora et al. [8]. 43

8 Multi-core and heterogeneous processors significantly increase the complexity of software design because they change the hardware/software interface. Multi-core processors require applications to explicitly coordinate among multiple instances of the same interface. Heterogeneous processors require applications to coordinate among multiple *different* interfaces. 52

9 Scaling of a GPU radix sort algorithm from Merril et al [104]. 73

10 CUDA source code for the inner loop of PNS. CUDA requires applications to be manually partitioned into GPU kernels and C++ statements that are executed sequentially. Harmony treats all kernels, C++ statements, and memory transfer operations as encapsulated kernels that execute out of order subject to data and control dependences. . . 80

11 The control flow graph for the PNS application. This is the representation that Harmony programs are compiled into before being executed. 81

12 The data flow graph for one iteration through the PNS loop. The black nodes represent kernels and the blue nodes represent variables that are managed by the runtime. Edges represent data dependences. 82

13 The data flow graph for two iteration through the PNS loop. Kernels from the second iteration have been invoked speculatively and red variables have been renamed to eliminate false data dependences. . . 83

14 A possible schedule for two iterations of the PNS application on a system with two different GPUs and a CPU. The green kernels in the figure are from the second, speculative, iteration. Note that speculation significantly increases the parallelism in the application. It is also worthwhile to note that the same kernel can have different execution times on different processors. 84

15 Common transformations on loop structures in the Harmony Kernel IR. 108

16 Variable assignment allows the compiler to trade-off space requirements and the amount of kernel parallelism in a Harmony program. 112

17 Navigating the space between common subexpression elimination and rematerialization allows additional storage to be allocated to hold intermediate results, or the same values to be recomputed multiple times. 114

18 Variable coalescing reduces the overhead of bounds checking, and potentially improves the efficiency of a series of copy operations via batching. 115

19 An example of kernel fusion, three simple kernels are merged into one larger kernel and their dependences are combined. 117

20 Statically partitioning the kernel data-flow graph allows heavy-weight partitioning techniques to optimize for load balance (middle section), memory traffic (right section), or other metrics without incurring the overheads of an NP hard problem at runtime. 120

21 Hardware branch predictors typically require fixed table sizes. Software implementations may dynamically grow the size of the table to fit the demands of the application. Different applications have significantly different demands on the table. 122

22 The difficulty of predicting branches remains relatively constant when increasing the granularity of kernels. 123

23 Variable renaming duplicates variables with a copy-on-write mechanism to eliminate false data dependences. 126

24 This figure shows examples of applications that experience a speed-up due to increased parallelism as a larger pool of memory is made available for renaming. 127

25 An overview of the translation process from PTX to multi-threaded architectures. 135

26 PTX thread hierarchy. 139

27 Example of PTX Barrier Conversion. Blue edges are branch targets and black edges are fall-through targets. The left CFG is split at the barrier, all live registers are spilled before the split point, a scheduler block is inserted to direct threads to the resume point, and all live registers are restored at the resume point. 144

28 Upper bound on kernel-level-parallelism in Harmony applications. The blue line represents KLP when variable renaming is not allowed. The green line represents KLP with renaming enabled. 153

29 The google protocol buffer specification of the Harmony binary format. Messages are encoded and serialized into a binary file. 155

30 An overview of the implementation of the Harmony Runtime. 158

31 The breakdown of time spent in each component of the runtime. . . . 159

32 KLP Predicted vs Measured Scaling (4 CPUs) 161

33 Scaling Without Speculation 162

34 Performance Improvement With Varying Speculative Depth 163

35 Memory throughput using different access patterns, all executing on a CPU. Note that strided accesses, which result in efficient coalesced accesses on GPUs, are inefficient on the CPU. 164

36 The throughput of atomic operations decreases as the number of host worker threads increases. The overhead of using locks only introduces at worst 20x the overhead over lock-less operations. 166

37 The overheads of context switches are primarily due to spilling live registers. Applications with few live variables are likely to perform well on CPUs. 167

38 CPUs are more efficient when performing integer operations. 168

39 Special instruction throughput. Texture operations are significantly slower without hardware support. 169

40 Performance of the same applications with different levels of optimization. The overheads of performing optimizations at runtime must be weighed against the speedup provided by executed more optimized code. The best optimization level is application dependent. 172

41 The breakdown of time spent in different Ocelot modules. x86 code generation is the most significant source of overheads. 173

42 The scaling of applications from the Parboil benchmark suite on 4 cores and 2-way simultaneous multi-threading. 174

43 The change in execution time of a CUDA benchmark with varying subkernel sizes. 181

44 Factor loadings for two machine principal components. PC0 (black) corresponds to single core performance, while PC1 (white) corresponds to multi-core throughput. 192

45 The machine principal components. GPUs have high core counts and slow SIMD cores while CPUs have fewer, but faster, cores. 193

46 Factor loadings for the five application principal components. A factor loading closer to +-1 indicates a higher influence on the principal component. 195

47 This plot compares MIMD to SIMD parallelism. It should be clear that these metrics are completely independent for this set of applications; the fact that an application can be easily mapped onto a SIMD processor says nothing about its suitability for a multi-core system. A complementary strategy may be necessary that considers both styles of parallelism when designing new applications. 196

48 A comparison between Control Flow Divergence and Data dependences/Sharing. Excluding the hotspot applications, applications with more uniform control flow exhibit a greater degree of data sharing among threads. Well structured algorithms may be more scalable on GPU-like architectures that benefit from low control flow divergence and include mechanisms for fine grained inter-thread communication. 199

49 This figure shows the effect of increased problem size on the Memory Intensity of the Nbody and Hotspot applications. While this relationship probably will not hold in general, it demonstrates the usefulness of our methodology for characterizing the behavior of individual applications. We had originally expected these applications to become more memory intensive with an increased problem size; they actually become more compute intensive. This figure is also useful as a sanity check for our analysis, it correctly identifies the Nbody examples with higher body counts as having a larger problem size. 200

50 Predicted execution times for the 280GTX using only static data. The left 12 applications are used to train the model and the predictions are made for the rightmost 13 applications. 203

51 Predicted execution times for the 8800GTX using only static data and all other GPUs to train the model. Black indicates the measured time, and gray is the predicted time. 204

52 Predicted execution times for the AMD Phenom processor using the Atom and Nehalem chips for training. 205

53 Predicted execution times for the 280GTX using all of the other processors for training. This is the least accurate model; it demonstrates the need for separate models for GPU and CPU architectures. 206

54 The skeleton CUDA implementation of cross product. 218

55 The major components in the RedFox compiler. The Datalog^{LB} language frontend converts programs into an AST representation. Datalog^{LB} operations are converted into a series of RA operations. These are converted one at a time into equivalent PTX kernels that are embedded in a Harmony binary and executed by the Harmony runtime and Ocelot dynamic compiler. 219

56 An example of the Datalog^{LB} selection $\rightarrow$ join benchmark. 222

Glossary

API Application Programming Interface. 159

ASIC Application Specific Integrated Circuit. 22, 54, 91, 92, 97

AST Abstract Syntax Tree. 219

BB Basic Block. 70, 71

BSP Bulk-Synchronous Parallel. 143

CFG Control-Flow Graph. 62, 70, 71, 79, 86, 99, 108, 109, 118, 156

CODEC Compressor-Decompressor. 97

CPU Central Processing Unit. 211

CSE Common Sub-Expression Elimination. 113, 114

CTA Cooperative Thread Array. 214–216

DFG Data-Flow Graph. 27, 71

DMA Direct Memory Access. 91, 95

DSP Digital Signal Processor. 54

FFT Fast Fourier Transform. 26

FIFO First-in First-out. 86

FPGA Field Programmable Gate Array. 44, 49–51, 95–97, 103

GB Giga-Byte. 99

GPU Graphics Processing Unit. 54, 92, 211

HIR Harmony Internal Representation. 36, 37

ILP Instruction Level Parallelism. 21, 70, 106

IR Internal Representation. 53, 118, 121

ISA Instruction Set Architecture. 27, 62, 92

KLP Kernel Level Parallelism. 31, 100, 110

LLVM Low Level Virtual Machine. 119

LUT Look-Up Table. 96, 103

MLP Memory Level Parallelism. 106

OOO Out of Order. 22, 23, 27, 31, 53, 59, 61, 85, 88, 92, 96, 97, 100, 102, 106, 112, 119, 126, 225

OS Operating System. 22

PGAS Partitioned Global Address Space. 66, 100

PTX Parallel Thread eXecution Instruction Set Architecture. 50, 119, 136, 138–140, 143, 146–149, 152, 157, 159, 164, 165, 168, 172, 184–186, 189, 192, 209, 214, 218, 220, 223, 227

RA Relational Algebra. 79, 209, 211–217, 220, 221, 223

RISC Reduced Instruction Set Computer. 27, 64, 97

ROB Reorder Buffer. 86

SIMD Single Instruction Multiple Data. 96

SOC System on Chip. 85

SPMD Single-Program Multiple-Data. 76

SSA Static Single Assignment. 38

VLSI Very Large Scale Integration. 21, 41

SUMMARY

The emergence of heterogeneous and many-core architectures presents a unique opportunity to deliver order of magnitude performance increases to high performance applications by matching certain classes of algorithms to specifically tailored architectures. However, their ubiquitous adoption has been limited by a lack of programming models and management frameworks designed to reduce the high degree of complexity of software development inherent to heterogeneous architectures. This dissertation introduces Harmony, an execution model for heterogeneous systems that draws heavily from concepts and optimizations used in processor micro-architecture to provide: (1) semantics for simplifying heterogeneity management, (2) dynamic scheduling of compute intensive kernels to heterogeneous processor resources, and (3) online monitoring driven performance optimization for heterogeneous many core systems. This work focuses on simplifying development and ensuring binary portability and scalability across system configurations and sizes.

CHAPTER I

INTRODUCTION

1.1 Performance Scaling on Heterogeneous Processors

The emergence of domain specific accelerators as a means to continue to deliver exponential performance gains from microprocessor systems has not come without a cost. The major results of decades leading up to this point have driven performance forward with a combination of manufacturing technology intended to reduce component cost, circuit design intended to increase clock frequencies, and processor architectures designed to to exploit Instruction Level Parallelism (ILP). The recent loss of aggressive threshold voltage scaling coupled to the increasing complexity of circuit pipelining and diminishing returns from ILP have ended the performance scaling of single-core processors and have driven modern designs towards explicitly parallel and specialized accelerators out of necessity, not out of choice. Although this switch has allowed performance scaling to continue, it comes at the cost of software development and complexity, where existing applications must be broken into structured and explicitly parallel kernels that can leverage highly parallel accelerator hardware, and at the same time must manage work distribution across accelerators with different performance characteristics and micro-architectures.

Technology scaling and Very Large Scale Integration (VLSI) have enabled first million, and eventually billions of transistors to be packed into a single processor. The design of these processors has been driven by the computational requirements of a variety of application domains, many of which still remain beyond the reach of modern systems despite decades of exponential increases in performance. As additional hardware resources have become available, processor architects and circuit

designers have largely hidden these resources using techniques like pipelining and Out of Order (OOO) execution that enable the use of many resources in parallel without changing the appearance that instructions are executed sequentially, one at a time. Unfortunately, physical limitations have ended the effectiveness of these techniques, resulting in a flattening of clock frequencies to control excessive power dissipation and diminishing returns from OOO execution. Technology scaling continues to provide additional hardware resources, and a great deal of recent research has been concerned with addressing the problem of exposing these resources to software that can effectively use them in parallel.

Multi-core processors are examples of this trend. They simply replicate a single general purpose core multiple times and delegate the management of each core to the Operating System (OS). Domain specific accelerators are another example where a single processor is composed of an Application Specific Integrated Circuit (ASIC) that is highly optimized to perform a single application. Heterogeneous systems may include several ASICs, and provide performance and efficiency improvements by delegating tasks to the most efficient ASIC, which can be several orders of magnitude more efficient than general purpose cores [63].

A primary challenge of using multi-core processors and domain specific accelerators is designing software that can effectively use them. Multi-core requires the design of explicitly parallel applications or intelligent auto-parallelizing compilers, each of which have had limited success to date. Accelerators may require re-writing an application by identifying operations that are suitable for mapping, and then using new formats for exchanging data with the accelerator, and starting or synchronizing computation on it. The lack of general purpose support typically complicates this problem further in cases where future processors may include different accelerators or similar accelerators with different interfaces.

This dissertation is concerned with the problem of managing work distribution

across heterogeneous processors with different capabilities and performance characteristics. It builds on existing work on the design of OOO and super-scalar processors, by extracting a core set of abstractions, composing them together to form a new execution model, and applying this model to scheduling application kernels on heterogeneous multi-core processors.

1.2 Execution Models

Since the core contribution of this dissertation is an execution model, it is useful to spend some time up-front to define execution models and review their recent evolution.

Execution models exist due to a bipolar rift between the goals of hardware and software. Software systems and programming models are designed for productivity. They are tasked with expressing algorithmic solutions to important problems in computing. Conversely, processor architecture, VLSI circuits, and semiconductor devices are designed for efficiency and reliability. They are constrained by the properties of physical devices. Execution models sit in the middle. They define a core set of abstractions that can both be used to express applications and also be efficiently mapped onto hardware. Whether they were designed explicit or arose naturally out of necessity is debatable. However, they exist in some form in all modern computer systems.

Instruction sets are probably the most well-known instantiations of execution models, although there is a wide variety of other examples [15, 82, 108, 144, 148]. They express applications in terms of operations (instructions) that transform data stored in registers or memory subject to dependences that restrict the order of their execution. During compilation, applications written in high level programming languages are mapped onto the execution model. At this point, static or dynamic optimizations can be applied to the execution model abstractions. Finally, the abstractions in

the execution model are **mapped** onto hardware devices (such as processors, memory banks, or functional units) and **scheduled** in time subject to their dependences. These processes of mapping and scheduling are common across almost all existing execution models.

Until the recent transition to parallel and heterogeneous processors, instruction sets have been effective execution models. This is evident in their widespread adoption and deployment in both specialized and commodity systems. However, there is no clear mapping from these models onto multiple heterogeneous cores that compose modern systems.

The core observation of this thesis is that instruction sets already have mechanisms for addressing with heterogeneity and parallelism, albeit at the functional unit level rather than the core level. There are two primary limitations preventing these techniques from being applied at a higher level:

1. The problem of mapping large sets of instructions from existing applications onto specialized cores is considered intractable.

2. The problem of partitioning an application into subsets of instructions with sufficient parallelism to leverage a highly-parallel processor is unsolved.

Harmony offers a solution to these problems by recasting applications in terms of compute kernels with explicit control and data dependences. Once in this form, existing solutions for mapping and scheduling can be applied by treating kernels like instructions and heterogeneous cores like functional units.

1.3 Relating Structure and Complexity

There is an overarching theme in this work that bears treatment up-front concerning the relationship between structured and irregular parallelism, and their impact on design complexity. The general philosophy is that as application parallelism becomes

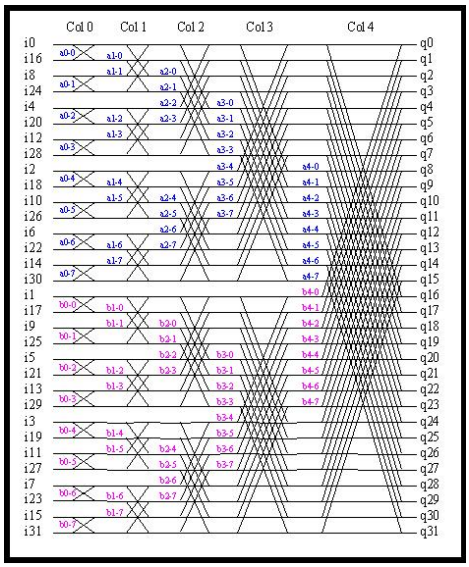

Figure 1: The data dependency graph for an FFT kernel.

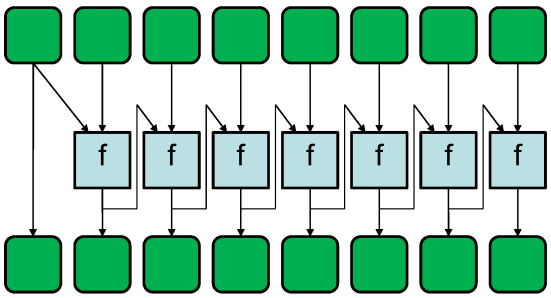

Figure 2: The data dependency graph for a simple vector addition kernel.

more irregular, dynamic and runtime techniques become more effective at discovering parallelism and distributing work. However, these techniques also become less effective as the degree of parallelism increases due to the eventual dominance of runtime overheads.

A simple measure of regularity can be obtained by decomposing an entire computation into a graph of control and data dependent operations and analyzing the structure of the graph. It is asserted that notions of structure and regularity are captured in the dependency information in this graph, which also obviously captures parallelism. One measurement that potentially captures structure is the degree of vertices in such a dependency graph. For evidence of this statement, we can consider

the dependency graphs of computations that are generally considered structured, such as the cooley-tukey Fast Fourier Transform (FFT) algorithm shown in Figure 1, or the simple data-parallel vector addition in Figure 2. Note that the degree of vertices (which represent operations) in these graphs is highly regular.

As the amount of variance in the vertex degree increases, the graph becomes composed of a mixture of operations that have few dependences and operations that gate the execution of many other operations. This makes dependences increasingly complex, less regular, and more difficult to deal with explicitly via common forms of synchronization such as barriers or reductions, where it is sufficient to add bulk synchronization at regular intervals after a group of independent operations. Runtime techniques excel at these unstructured problems by tracking all dependences for a limited window of active operations. However, as the number of operations tracked by the window increases, the overhead of maintaining and updating a list of dependences grows super-linearly and eventually becomes a serial bottleneck. In these cases it generally becomes more advantageous to recast the application in terms of high level groups of low level structured operations. Specific instances of this phenomenon can be observed in the migration from complex circuit operations to general purpose instructions, or again from instructions to compute kernels.

This dissertation introduces a new execution model, Harmony, that casts applications as flow graphs of compute kernels. It relies on scheduling and optimizations techniques that fall into dynamic and runtime categories. As such, it is most suited to problems with a limited degree of possibly unstructured parallelism. In this case, the problem will be mapping a set of tasks with arbitrary dependences onto a set of available accelerators.

1.4 Roots in Out of Order Execution

Throughout this dissertation, it will become apparent that many of the core concepts have direct analogs in common Instruction Set Architecture (ISA) abstractions. This section provides a brief overview of these abstractions.

1.4.1 Instruction dependences

Instruction in many Reduced Instruction Set Computer (RISC) ISAs are specified sequentially and perform basic units of work on registers. The sequential ordering creates a set of implied dependences between instructions that modify the same registers. Each may be a true dependency, where data actually flows from the result of one instruction into the input of a later instruction, or it can be a false dependency when multiple instructions share the same register, but do not share data. Extending this concept, a series of arithmetic or logic instructions can be represented as a directed acyclic graph, where nodes specify instructions and edges represent data dependences. This is referred to as a Data-Flow Graph (DFG). The correct execution of these instructions can be satisfied by any ordering that obeys the constraints of this graph, giving rise to OOO execution as a method for executing multiple instructions in parallel.

1.4.2 Dependency-Driven Instruction Scheduling

As instructions are fetched and decoded, they are stored in an issue queue before they can begin execution. Although the actual hardware mechanism may be implemented in multiple ways, instructions must wait to begin execution until other instructions that they depend on have generated their results. Note that false dependences can be eliminated using register-renaming, which uses temporary storage to pass a value between instructions with true dependences without creating a register resource conflict. Finally, ready instructions can begin execution on functional units subject to their availability. Various algorithms and heuristics are employed to determine which

instructions are given preference during scheduling.

1.4.3 Branch Prediction

Other than data dependences between instructions, it is also possible for the execution of instructions to depend on the outcome of a branch. In this case, branch prediction, a form of speculative execution, can be used to guess the expected outcome and next target of a branch instruction and remove the dependency. If the prediction is later determined to be incorrect, the system is reverted back to a previous state and the correct instructions are executed. If-conversion is a complementary technique that converts control dependences into data dependences by executing all targets of the branch and selecting the appropriate results. Additionally, some analyses can identify regions of code following a branch as control-independent, meaning that they will be eventually executed regardless of the outcome of the branch. In these cases, control-independent instructions can be executed without waiting for the outcome of the next branch.

Readers who have had exposure to these topics should find the abstractions in the Harmony execution model that is presented in this dissertation familiar.

1.5 Managing Dynamic Overhead

The systems and analyses that are employed in this dissertation to extract parallelism invariably come at a resource cost (where cost measured in terms of area, power, or complexity). In the case of single-chip processors, this cost is manifested in the complex instruction caches that fetch multiple lines containing many instructions in a single cycle, multi-ported register files that must meet the bandwidth constraints of supplying operands to several instructions each cycle, the large physical register files and register alias tables needed to eliminate false data dependences by creating scratch-pad storage for temporary values, and multi-level branch predictors with large history tables, among others. This phenomenon occurs on different scales as

well. The runtime decision models introduced later in this thesis can provide additional performance by exposing additional parallelism, hiding latency, and steering computations to more efficient hardware. However, these decision models consume system resources, and at some point, the resource cost needed to exploit more dynamic behavior of an application begins to outweigh the performance improvement.

This section is meant to serve as a warning to the reader of the dangers of blindly applying the techniques presented in this dissertation as blunt instruments that are used to hammer out additional performance. Several historical examples [86, 128] along with cost and benefit studies [118] should stand as grave reminders of the consequences of taking these techniques too far. Any use of this model should be accompanied by a careful evaluation of the expected costs and benefits to avoid the problem of diminishing returns. Once the costs begin to significantly outweigh the performance improvements, it is usually a more efficient use of resources to move these techniques up a level of granularity and apply more regular, explicitly parallel, models at the current level. This general philosophy can be seen in the move from complex custom circuits to forms with regular structure such as Kogge-Stone parallel prefix-sum adders [87] and Wallace-tree multipliers [152] that implement function units. It can also be seen in the migration from complex out-of-order micro-architectures to tiled arrays of highly multi-threaded cores with modest superscalar width in single-chip processors and accelerators.

1.6 Towards a Generic Execution Model

This thesis identifies common abstractions from several existing systems and composes a generic execution model that can be applied to systems at various granularities. A central theme of this work is that dynamic methods can be used to automatically extract irregular and unstructured parallelism from a sequential series of operations and distribute them on a set of processor resources with heterogeneous performance

characteristics and capabilities. Although the examples presented in the following chapters focus exclusively on showing how this model can be applied to the problem of distributing and scheduling tasks on accelerator boards in a system containing several processors, each with a specialized micro-architecture, the model itself is more generic, and the abstractions are designed to have broader application.

There are several similarities between specialized accelerators and functional units that suggest that they may be able to be reasoned about using the same abstractions. They each perform a specific operation, consuming inputs and producing outputs. They are specialized for one operation, such as floating point multiplication, or a single class of operations, such as video decoding or encryption. An entire application is composed of a series of data-dependent chains of instructions as well as branches that determine the next series of instructions to execute. Similarly, many applications for accelerator based systems include a sequential flow of control that issues a series of compute kernels. The later chapters of this dissertation leverage these similarities to provide generic abstractions for branches, kernels, data values, and associated analyses that are applicable to both classes of systems.

There is an implicit assumption that technology and market factors will result in a change in the number and configuration of processing elements in a system. Technology scaling has historically enabled the use of many functional units, and it is likely that the same force will enable the integration of many accelerators on the same processor. Applications that expect to transparently improve performance as more resources become available require an automatic mapping function from kernels and instructions to accelerators and functional units. Additionally, applications that are not highly optimized benefit from this dynamic mapping because it performs some of the heavily lifting of scheduling and resource allocation dynamically to every application. Finally, scheduling and optimization that is performed at runtime has certain advantages over equivalent static techniques that do not have information to

data-dependent dynamic program information, such as the outcomes of control decisions and branches. It will be shown that many of the existing dynamic optimizations that are applied to instructions in OOO processors can also be applied to accelerator systems through the use of common abstractions.

Although there are many similarities between accelerator systems and OOO processors, there are also a few caveats. Accelerators are typically more capable than functional units. Rather than being restricted to a few canonical logical and arithmetic operations on binary values, many accelerators are Turing complete; they are able to execute general purpose applications with varying degrees of efficiency, depending on the application. Additionally, it is usually the case that the expected execution time of a kernel on an accelerator is not known statically; the problem of determining the execution time of all possible kernels on all possible hardware is not feasible, whereas it is relatively simple to statically determine the latency of functional units. Even with this difference, instructions are still captured by the model with the constraints that they are restricted to specific functional units, and their performance is statically known.

1.7 Contributions

The principle contribution of this dissertation is the formulation of abstractions and techniques traditionally used by OOO super-scalar processors into a generic execution model that can be applied to systems at multiple levels of granularity.

In addition to this, it makes the following contributions:

- It provides a definition for Kernel Level Parallelism (KLP) and an empirical evaluation of KLP in Harmony and CUDA applications.
- It describes the design of a prototype implementation of the Harmony runtime.
- It introduces the concept of using dynamic binary translation to map the same

bulk-synchronous binary representation of a program onto processors with different micro-architectures.

- It defines the Subkernel program structure, and demonstrates its effectiveness in aiding bulk-synchronous dynamic compilation.
- It proposes a statistical framework for predicting the performance of compute kernels to aid in scheduling.
- It evaluates a Datalog language front-end for the Harmony execution model.

1.8 Thesis Organization

This dissertation begins with an executive summary of the Harmony execution model that focuses on the mapping of an example application onto the model and its execution on a hypothetical machine. This followed by a breadth first survey of Harmony, moving from related work, through the model abstractions, and ending with possible optimizations that can be applied to the model. The final section includes several deep dive studies into important aspects of the execution or optimization of Harmony applications. Finally, the dissertation concludes with a summary and suggested directions for future work.

A brief overview of each chapter is listed as follows. Chapter 3 describes the current landscape of related work and puts this thesis into perspective. Chapter 4 describes the abstractions used in the Harmony execution model, several common ways to analyze them, and an example of their composition into a complete application. Chapter 5 takes these abstractions and shows how they can be applied at multiple granularities. It covers out of order execution in processor micro-architecture, focuses on distributing compute kernels onto multiple accelerator boards, and briefly outlines its potential future use for distributing large-scale operations on partitions of a heterogeneous cluster. Chapter 6 explores optimizations that can be applied

statically by a compiler or dynamically by a runtime to the model. Several existing optimizations are recast in terms of the model abstractions and others are introduced that only make sense at the level of systems of accelerators. Chapter 7 describes the execution model that is used to execute an operation on a single processor. It shows how this model can be translated to different types of processors. Chapter 8 explores several aspects of the model in detailed case studies. Finally, Chapter 9 concludes with insights drawn from the case studies and offers suggestions for follow-on work.

CHAPTER II

AN EXECUTIVE SUMMARY

Harmony applications consist of variables, control decisions, and compute kernels. Variables are named objects that store data; they may be single integer values, arrays, or complex data structures such as tree or hash tables. Applications are executed sequentially, one compute kernel or control decision at a time. Compute kernels consume input variables and produce or update output variables. Control decisions consume input variables and change the flow of control through a Harmony application.

Applications in this form are mapped onto an abstract model of a machine, which is composed of memory spaces, processors, a central control unit, and explicit communication channels. Variables are scheduled onto memory spaces and compute kernels are scheduled onto processors. Communication channels constrain data movement between connected memory regions; they also constrain kernel execution to processors that are connected with memory regions with copies of their input and output variables. Processors are assumed to be heterogeneous, and kernels are executed on processors using dynamic compilation or emulation.

Consider the expression of a well-known algorithm, such as matrix multiplication, in terms of these abstractions. Matrix multiplication is ubiquitous enough to be familiar to many readers, so it is used as an example.

2.1 A Sample Machine Model

A machine is targeted with three processors {P1, P2, P3} and three memory spaces {M1, M2, M3}. Figure 6 depicts the connections from memory spaces to processors, as well as the connections between memory spaces. In this configuration, P1 can

```
   Input: input blocked matrix A
   Input: input blocked matrix B
   Output: output blocked matrix C
1  foreach y in 0..2 do
2      foreach x in 0..2 do
3          Variable sum := Block(0)
4          foreach k in 0..2 do
5              Variable* a := getVariableAtIndex(k, y, A)
6              Variable* b := getVariableAtIndex(x, k, B)
7              Variable temp := multiply(a, b)
8              sum := add(sum, temp)
```

Algorithm 1: An example of 3x3 blocked matrix multiply in terms of Harmony kernels.

execute kernels and also acts as the central control unit that executes the Harmony runtime. Its corresponding memory space (M1) is accessible to P1 as well as the other memory spaces (M2 and M3). However, the other processors (P2 and P3), only have direct access to their corresponding memory spaces (M2 and M3). This configuration represents a common accelerator system where P1 represents the CPU that can orchestrate data communications through the entire system and the remaining processors represent accelerators with private memories. In this example, all processors are assumed to have different performance characteristics. Specifically, assume that P1 takes 10 units of time to multiply a matrix block, P2 takes 4 units of time, and P3 takes 2 units of time. P1 needs 5 units of time to add a matrix block, but both P2 and P3 can perform the same operation in 2 units of time. All processors perform all other operations including memory transfers in 1 unit of time.

2.2 Matrix Multiplication in Harmony

This example assumes that several primitive kernels are available as building blocks. These include a dense matrix multiply kernel (K_{GEMM}), an element-wise matrix sum kernel (K_{sum}), an indexing kernel (K_{index}), a compare value kernel ($K_{compare}$), and an increment value kernel ($K_{increment}$). A single compare-and-branch control decision

```
Entry:
  K1   Variable y     = Value(0)
  K2   Variable limit = Value(2)

Loop0:
  K3   Variable x   = Value(0)

Loop1:
  K4   Variable k   = Value(0)
  K5   Variable sum = Block(xDim, yDim, 0)

Loop2:
  K6   output Variable blockA = K_index(input k, input y, input A)
  K7   output Variable blockB = K_index(input x, input k, input B)
  K8   output Variable temp   = K_GEMM(input blockA, input blockB)
  K9   output Variable sum    = K_sum(input sum, input temp)

  K10  output k = K_increment(input k)
  K11  output Variable condition2 = K_compare(input k, input limit)
  K12  if(not condition2) goto Loop2

  K13  output x = K_increment(input x)
  K14  output Variable condition1 = K_compare(input x, input limit)
  K15  if(not condition1) goto Loop1

  K16  output y = K_increment(input y)
  K17  output Variable condition0 = K_compare(input y, input limit)
  K18  if(not condition0) goto Loop0

Exit:
  K19  exit
```

Figure 3: An example of 3x3 blocked matrix multiply in the Harmony IR.

(C_{branch}) is included as well. These primitive kernels are combined to implement blocked matrix multiply.

The application consumes a set of variables representing the blocked regions of two input matrices, $\{A_{ij}\}$ and $\{B_{ij}\}$ respectively. It produces the blocked regions of an output matrix $\{C_{ij}\}$. Although a more realistic application would allow the matrix sizes to be specified as input parameters, consider a simple case where all matrices are composed of $3x3$ sets of blocks. The application takes the logical form of Algorithm 1.

It consists of three nested for-loops. The first loop iterates over the columns in C, the second loop iterators over the rows in C, and the final loop co-iterates over the corresponding rows of A and columns of B to form a dot product. During the compilation phase of the program, this syntactic form is lowered into a sequence of kernels and control decisions by a high level compiler's language front-end. One possible form is shown in Figure 3. This form is referred to as the Harmony Internal Representation (HIR) and it is the form that is executed by the Harmony runtime. Note that all variables are tagged with explicit input and output access modes. These

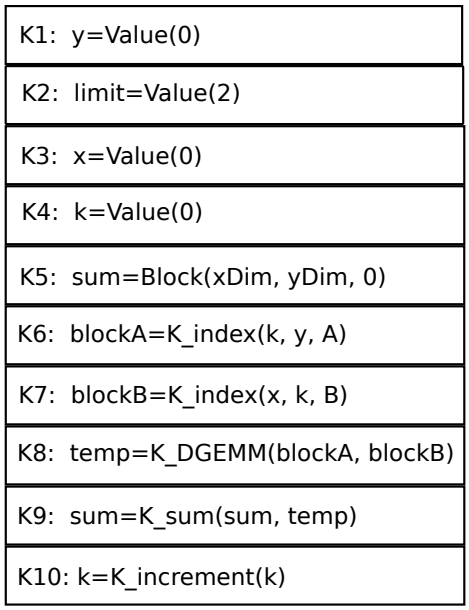

Figure 4: The initial state of the fetch buffer for the Harmony 3x3 matrix multilpy example.

enable the runtime to determine data dependences through memory. A high level compiler will serialize the HIR into an executable binary format.

2.3 Execution with the Harmony Runtime

When this program is executed, the Harmony runtime begins fetching kernels from the HIR binary. The fetch logic in the runtime is free to fetch individual kernels at a time, entire basic blocks, or multiple basic blocks using control flow speculation. In this example, assume aggressive fetch logic that is able to fetch multiple basic blocks speculatively and that employs a predictor that initially assumes taken for backwards control decisions. For this example, the fetch logic will predict the first three iterations of each loop correctly and miss-predict the last iteration. Speculative kernels are explicitly marked by the frontend. Fetch continues until a buffer of fetched kernels is filled. In this example, assume that the fetch buffer can hold ten kernels at a time. During the first fetch operation, the initialization kernels and one iteration of the inner loop will be pushed into the fetch buffer. This example is shown in Figure 4.

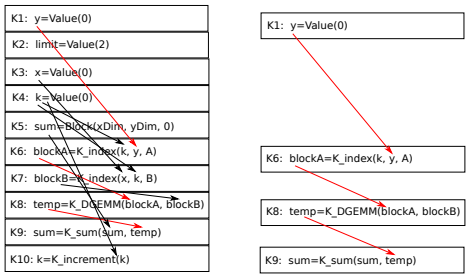

Figure 5: The initial dataflow graph for the Harmony 3x3 matrix multilpy example. The right side of the figure shows the critical path.

Kernels that have been fetched and buffered are available to the next stage of the runtime, which performs scheduling and resource allocation. During scheduling, the runtime's scheduling logic consults a performance predictor to determine the expected execution time of each kernel on each processor in the system. The predictor also estimates the time for memory transfers. For simplicity, this example assumes that these predictions are accurate. However, in a real system, the performance model will be trained over time as the program executes and will include some degree of error.

A first pass converts variable names into Static Single Assignment (SSA) form to eliminate false data dependencies. The scheduling pass itself identifies the critical dataflow path through the program, shown in Figure 5 to be (K1, K6, K8, K9), which is scheduled first using bottom-up list scheduling. Note that the choice of scheduling algorithm is left up to the implementation of Harmony runtime. Based on the costs given in the beginning of the example, K9 is scheduled first on P3. Next, K8 is scheduled on P2 because it has the lowest total execution time; even though this incurs an extra move operation from M3 to M2, the scheduler does not backtrack. K6 and K1 incur an equal cost on all processors, so they are also scheduled on P2 to avoid data copies. Once the first critical path is scheduled, the remaining kernels are scheduled by their criticality. The final schedule is shown in Figure 6. Note that indexing kernels are treated specially by the scheduler because they do not necessarily move data. They only choose one out of a set of input variables. So even though

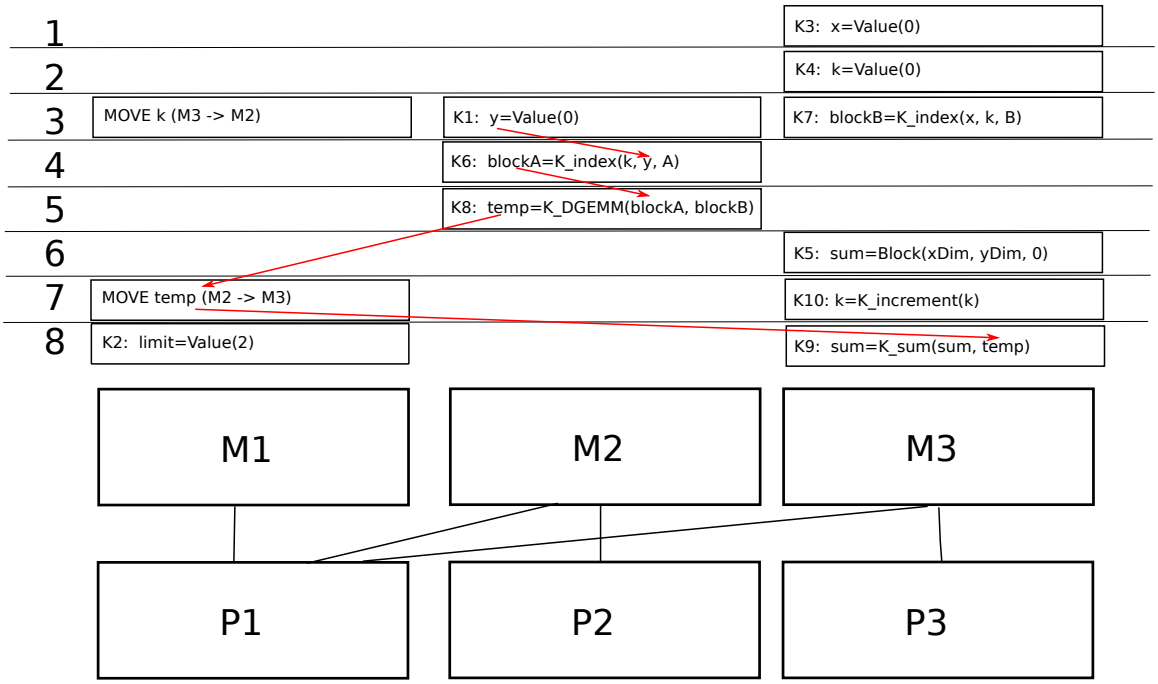

Figure 6: The initial kernel schedule for the Harmony 3x3 matrix multilpy example. The critical path is shown in red.

they specify that they will access all of the blocks in matrix A, B, or C, the scheduler only models the cost of executing the indexing kernel and moving a single block if the input variables are in a different memory space than the user of the output variable.

Once kernels and data movement operations have been scheduled, they are queued for execution on processors. If a native binary is not available for a processor, a dynamic kernel compiler is invoked. The runtime tracks kernels as they begin execution until they complete. It frees memory for variables that are never accessed again. It also records the execution time and other statistics of each completed kernel and uses this information to update the branch predictor and kernel execution time predictor. While scheduled kernels are executing, the fetch logic will refill the fetch buffer and the process will repeat until the program terminates.

Now, armed with a high-level understanding of the Harmony abstractions and an example of their runtime mapping onto a hypothetical machine, the next sections begin exploring these concepts in greater detail and put them in the context of related

work.

CHAPTER III

ORIGIN OF THE PROBLEM AND CURRENT LANDSCAPE OF RELATED WORK

3.1 The Rise and Fall of General Purpose Computing

The driving forces of many-core scaling and heterogeneous acceleration have the potential to revolutionize general purpose computing. Recently, graphics accelerators from NVIDIA and AMD have been released with up to 4.14 TFlops of single precision throughput and 256GB/s of off-chip memory bandwidth [5, 111]. At the same time, Intel Research has demonstrated a 48-core tiled IA32 processor that can boot Linux [70]. These processors are the culmination of tremendous advances in semiconductor manufacturing, VLSI, circuit design, and micro-architecture. However, they are also a significant departure from the Von Neumann model and greatly increase the complexity of software design.

Though the evolution of the computing industry has included forays into parallel processing, vector architectures, and application specific accelerators, the last few decades have been dominated by general purpose computing and high volume, commodity processors. Commodity general purpose processors were successful because they were able to steadily increase performance while maintaining a static hardware-software interface. Applications could be written and compiled once, for a single ISA, and frequency and ILP scaling would ensure that new processors would not only correctly execute an application, they would do so with greater performance. However, as power constraints began to limit frequency scaling and the amount of available ILP in existing applications was exhausted, general purpose processors began to again move towards parallel and domain specific architectures.

3.2 The Many-Core and Heterogeneous Revolution

3.2.1 Many-Core Processors

Even before technology constraints forced the industry migration away from aggressive super scalar architectures, several research efforts began developing architectural solutions to the increased wire delay, limited off-chip bandwidth, growing power consumption even in the face of voltage scaling, and mounting design and verification complexity as Moore's Law scaling enabled the production of billion transistor processors [19]. These efforts spanned the spectrum from aggressive super-scalar designs to reconfigurable fabrics of parallel components. They were polarized between two opposing design philosophies. Some focused on working within the constraints of the existing hardware/software interface. These included advanced super-scalar designs that extended the upper bound on ILP from 2-4 to 16-32 [122], trace and multi-scalar architectures that attempted to execute bundles rather than individual instructions speculatively [134], culminating in compiler-assisted hardware transactional memory and speculative threading [65,133]. The alternative philosophy focused on technology driven designs that could refine or even redefine the hardware/software interface. Some prominent examples included processors built around simultaneous multi-threading [49], vector and SIMD architectures [88], chip-multiprocessors [115], and polymorphic and tiled architectures [20, 135, 151]. Multi-threaded and multi-core architectures retained a uniform ISA, but required applications to coordinate communication and synchronization across multiple threads. Ultimately, a hardware/software interface defined by a single processor with a uniform memory space and a static ISA became a liability and the general purpose computing industry moved to chip-multiprocessors and domain specific accelerators. One of the most daunting challenges in computing today is dealing with the fallout from this decision.

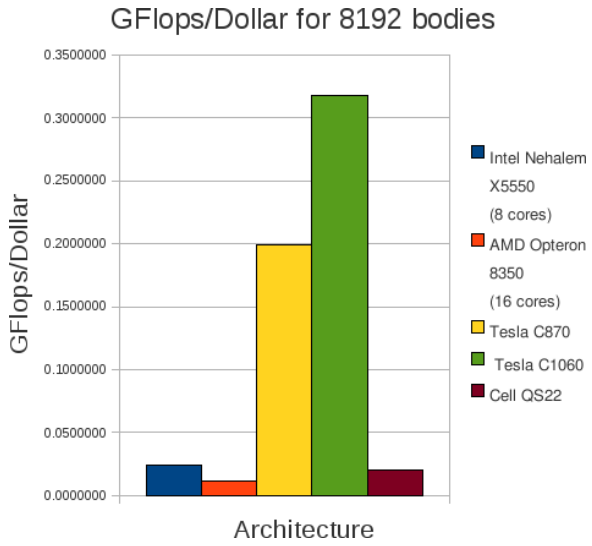

Figure 7: GFPLOPs per dollar for different multi-core platforms when running the Nbody application developed by Arora et al. [8].

3.2.2 Heterogeneous Systems

General purpose architectures inevitably sacrifice efficiency for programming simplicity and generality. Although the prohibitive cost of designing and manufacturing a processor limits the economic feasibility of ASICs, certain application domains that are limited by the efficiency and capability of general purpose processors can still support the design of custom architectures if they are driven by a large enough market. These conditions exist in domains such as computer graphics, signal processing, and embedded computing where accelerators have been designed that offer significant performance and efficiency advantages over general purpose architectures. Figure 7 compares the evolution of graphics accelerators and general purpose processors in terms of FLOPs/Watt. These architectures are not heterogeneous individually. However, the ability to integrate many processors together onto the same die presents the opportunity to compose truly heterogeneous system on chips, with different processors optimized for different classes of workloads. These architectures are becoming increasingly important as designs become more power constrained.

Heterogeneous processors can be grouped into three categories: heterogeneous

micro-architectures, ISA extensions, and heterogeneous cores. At the micro-architecture level, allocating hardware resources differently can affect the performance of certain classes of applications. For example, Najaf-abadi et al. propose a scheme for architecture contesting where several cores, each with a different micro-architecture, are included in a multi-core processor and used to accelerate workloads with different characteristics [106]. At the ISA level, it is possible to add additional instructions to the base ISA for accelerating domain specific operations. For example, SIMD extensions to the x86 instruction sets allow vector operations to be accelerated on some general purpose processors [125], and approximate transcendental and texture interpolation instructions are included in NVIDIA GPU ISAs [37]. Finally, cores with completely different ISAs and micro-architectures can be incorporated into the same multi-core processor. Wong et al. have prototyped a combined Intel GMA X4500 GPU and Intel IA32 CPU on the same Field Programmable Gate Array (FPGA) [153]. Recently released designs from Apple [7], Intel [74], and NVIDIA [113] along with announced projects from AMD [80] pair CPU and GPU cores together on the same processor. It should go without saying that both ISA extensions and heterogeneous cores require significant changes in the hardware/software interface as illustrated in Figure 8.

3.3 Programming Languages for Parallel and Heterogeneous Systems

Though the return to parallel and heterogeneous architectures has allowed performance scaling to continue, the accompanying hardware/software interface changes have ended the forward scalability and, in some cases, the forward compatibility of existing applications. These problems have been pushed higher up the stack, into compilers, runtime systems, programming languages, and even algorithms and data structures. The many-core revolution has forced programmers to address parallelism, and several industry and research efforts have developed implicitly and explicitly parallel models for multi-core programming. Beyond typical message passing and

threading models, streaming and bulk-synchronous-parallel languages have emerged as viable programming models for many-core processors. Several efforts have also addressed architecture heterogeneity with new language semantics and operating system and runtime support.

3.3.1 Implicitly and Explicitly Parallel Programming

Following the introduction of multi-core processors, the first languages and programming models to take advantage of the new capability were threading models that had previously been used to program multi-processor systems with shared memory [108] and shared nothing models like MPI that used message passing for communication on clusters [117]. In these explicitly parallel models, the programmer is responsible for launching threads, mapping them to cores, and coordinating inter-thread communication and synchronization. An alternative philosophy quickly emerged that relied on programmer inserted hints and intelligent compilers to automatically extract parallel regions from sequential applications and map them onto threads. Today, OpenMP [26] is probably the most popular realization of this philosophy. However, a number of research efforts identified several potential problems with these solutions to parallel programming. Most notably, 1) that programs written for a fixed number of processes in MPI or threads in Pthreads could not be scaled without modification to future processors with higher core counts [15], and 2) that the ability to perform operations with side-effects and unstructured, non-deterministic, synchronization and communication significantly increased the complexity of writing parallel applications [82]. Cilk [15], Intel TBB [35], and Ct [73] are programming languages that address the problem of scalability by expressing applications in terms of large groups of light-weight threads coupled with runtime components that provide low overhead synchronization and

communication operations. Alternatively, Charm++ [82] adopts a message passing model and focuses on the problem of complexity by requiring threads to execute encapsulated functions without side effects and explicitly specifying channels for communication. Finally, streaming and bulk-synchronous-parallel languages are emerging models for parallel programming that are significantly different from traditional threaded or shared memory models. They are discussed in more detail in the following sections.

3.3.2 Stream Programming

Stream computing is driven by a popular class of applications that operate on signals or series of data. Though there are many subtle variations for different languages, programs are typically composed of directed data-flow graphs of 'kernels' that operate on explicit input and output streams and perform some transformation on elements from the input streams to produce elements on the output streams. Prominent examples of streaming languages include StreamIt [144], the Stream Virtual Machine [90], and Brook [71]. Streaming languages have well defined semantics that ease their migration onto parallel architectures. In the simplest case, kernels can be processed concurrently by exploiting pipeline parallelism [143]. This technique has been used to map streaming languages to the IBM cell processor [159], multi-core x86 processors [89], NVIDIA GPUs [147], AMD GPUs [4], and several others [55]. Recently, compiler transformations have been proposed that map streaming languages onto vector SIMD processors [69]. Although streaming languages have significant advantages, some applications, particularly those with unstructured control and data flow, do not map well to the streaming model.

3.3.3 Bulk-Synchronous-Parallel Languages

Bulk-Synchronous-Parallel (BSP) programming is based around the idea that the cost of global synchronization will eventually limit the scalability of parallel applications.

Applications are composed of many parallel tasks that execute independently and can communicate via either message passing or shared memory. However, messages are only guaranteed to be received and shared memory is only guaranteed to be consistent after periodic global barriers. Applications are encouraged to launch as many tasks as possible between barriers to amortize the global synchronization overhead. As tasks cannot reliably communicate between barriers, there are no constraints on the scheduling of tasks; tasks can be executed in parallel on different processors, or serialized on the same processor and executed to completion without a context switch. Since the original formulation of the BSP model by Valiant [148], several industry initiatives have adopted variants for general purpose programming on GPUs. CUDA was the first example such example, released by NVIDIA in 2007 [110]. OpenCL followed with a specification in 2008 [57] and the first compilers began to emerge in late 2009 [6, 112]. Both CUDA and OpenCL express programs in terms of a series of compute kernels, which are composed of highly multi-threaded tasks (blocks in CUDA terminology). Tasks cannot synchronize within kernels. However, a shared memory space is made consistent across kernel launches, which are effectively global barriers. A significant addition to the BSP model made by OpenCL and CUDA is that tasks are not persistent across kernel launches. This limits the amount of state needed to represent a kernel; it is reduced to the total number of active tasks rather than the total number of tasks, which can be potentially overwhelming in cases where millions or billions of tasks are launched [42].

3.3.4 Heterogeneous-Aware Languages

Though there has been comparatively less work addressing heterogeneity than there has been on reducing the complexity and increasing the scalability of parallel applications, there are a few notable exceptions. Linderman et al. propose the Merge framework for writing applications for heterogeneous systems consisting of CPUs and

GPUs. Programs in Merge are expressed as a series of kernels with 'predicates', which are characteristics of the processors that the kernel can execute on. A Merge program is executed by a runtime that uses explicit dependences between kernels coupled with predicates to determine the set of possible mappings from active kernels to available processors and choose a good mapping. Merge offers one approach for handling micro-architecture hierarchy. However, it does not directly address systems where processors each have their own memory hierarchies. Fatahalian et al. address the problem of memory system heterogeneity with Sequoia [50], another runtime supported programming model that expresses an application as a tree of kernels with explicit data dependences. The structure of a Sequoia application allows it to be recursively partitioned into different levels of DRAM and cache in a complex memory hierarchy. A common feature of these approaches is the expression of an application in terms of kernels, and a runtime component that dynamically maps kernels to available hardware resources. The use of a runtime introduces the need to schedule kernels effectively given the fact that the same kernels can perform significantly better or worse on a given processor. Jimenez et al. explore several existing scheduling techniques in heterogeneous systems with GPUs and CPUs [77]. They conclude that predictive models that attempt determine the expected execution time of each kernel on each available processor are necessary for efficient kernel scheduling. Noticeably lacking from this body of work are the problems of 1) executing the same kernels on processors with different ISAs, 2) migrating from existing languages to these new frameworks, and 3) static and dynamic optimizations that can be performed on kernel representations of programs.

3.4 Dynamic and Heterogeneous Compilers

Even before the widespread adoption of truly heterogeneous systems, the use of different ISAs in competing commodity processors prevented applications from being

executed on multiple platforms and restricted the design of new micro-architectures. Dynamic compilation (also known as just-in-time compilation) was born out of this lack of application portability. Previous work in dynamic compilation has been concerned mainly with enabling portability and performance across Von-Neumann architectures with different ISAs. However, there is an independent body of work that addresses static compilation from bulk-synchronous-parallel languages to a variety of Von-Neumann, Vector, and even FPGA architectures. One of the major contributions of this work is to show that these complementary bodies of work can be combined to provide transparent portability of BSP languages across Von-Neumann, Vector, and FPGA architectures.

3.4.1 Dynamic Compilers

Though virtual instruction sets and interpreters have existed for decades, their poor performance has historically precluded their widespread adoption. In 1999, DEC released FX32!, a dynamic binary translator from x86 to the Alpha instruction set; FX32! allowed x86 applications to be executed on the 21364 series of Alpha processors [29]. Rather than translating an entire application at once and then executing it, FX32! translated a program as it executed by trapping into the compiler when untranslated code was encountered and saving the translated instructions in a translation cache. Following this project, several languages that were originally compiled into virtual ISAs and interpreted, for example, Java and CLR [103], moved to dynamic compilers that translated from virtual ISAs to the native ISA of a particular processor. Whereas the typical overheads of interpretation resulted in 40x or greater slowdowns over native versions of the same application, dynamic compilation and just-in-time translation made virtual ISAs competitive with native compilers. In fact, the ability to use runtime information to inform compiler optimizations spawned

several projects such as Dynamo [11] and Jalepeno/Jikes [12] that were able to outperform native compilers for some applications. The success of these projects lead researchers to explore problems that were even more ambitious, for example, Hydra [114] showed that dynamic compilation could be applied to VLIW architectures, and Transmeta introduced an entire line of VLIW processors coupled to a dynamic compiler that could execute native x86 applications [38]. In the context of this dissertation, the rapid development of dynamic compilation frameworks culminated in the release of the NVIDIA Parallel Thread eXecution Instruction Set Architecture (PTX) virtual ISA and dynamic compiler, which provided a low level ISA for expressing bulk-synchronous-parallel applications and a compiler for mapping this model onto multi-core vector processors (GPUs) [37, 60].

3.4.2 Heterogeneous Compilers

One of the primary limitations of heterogeneous systems is the existence of significantly different programming models and languages for different accelerators. Typically an application written using a single programming model will function on only a single type of accelerator, and it is usually not possible to even evaluate the suitability of the same programming model for multiple accelerators. However, there have recently been a few attempts to map parallel programming models typically used for multi-core CPUs to GPUs, for example source-to-source translators from OpenMP [94] and MPI [139] to CUDA have been proposed. Other research efforts have explored mapping GPU programming models to other processors. Stratton et al. have developed MCUDA, a source-to-source translator from CUDA to C that allows the same applications to execute on any CPU with a C compiler and any NVIDIA GPU with a CUDA compiler [138]. This work was extended by Papakonstantinou et al. in FCUDA to include a source-to-source compiler from CUDA to AutoPilot, which is a synthesizable subset of the C language that can be mapped onto FPGAs [119].

This line of work is very interesting because the performance of MCUDA and FCUDA are competitive with existing languages for FPGAs and multi-core CPUs. It suggests that the CUDA programming model is a good match for several classes of processors, not only GPUs. Furthermore, it suggests that it should be possible to develop a compiler with multiple back-ends that allows applications to be executed on systems with CPUs, GPUs, and FPGAs.

3.4.3 Optimizations for Heterogeneous Compilers

With the growing availability of CUDA compilers for different accelerators, it is becoming increasingly important to optimize their performance. Although some existing compiler optimizations can be applied directly to CUDA applications, for example, Murthy et al. have explored loop-unrolling [105], and Dominguez et al. have covered register allocation [48], the explicit notions of parallelism and synchronization create unexplored opportunities for parallel-aware optimizations. For example, Stratton et al. offer a preliminary exploration into the trade-offs between performing redundant computations in parallel and or assigning them to a single thread and then broadcasting the results to all other threads [137]. Another interesting opportunity presented by the CUDA execution model is the ability to map a SIMT (multi-threaded) execution model onto a SIMD (vector) machine. Fung et al. present a hardware optimization that dynamically regroups threads into vector bundles (warps) to improve the utilization of SIMD processors [51]. Tarjan et al. extend this mechanism to split warps on cache misses to hide memory latency [141]. These optimizations have great potential and yet have not been evaluated in CUDA compilers.

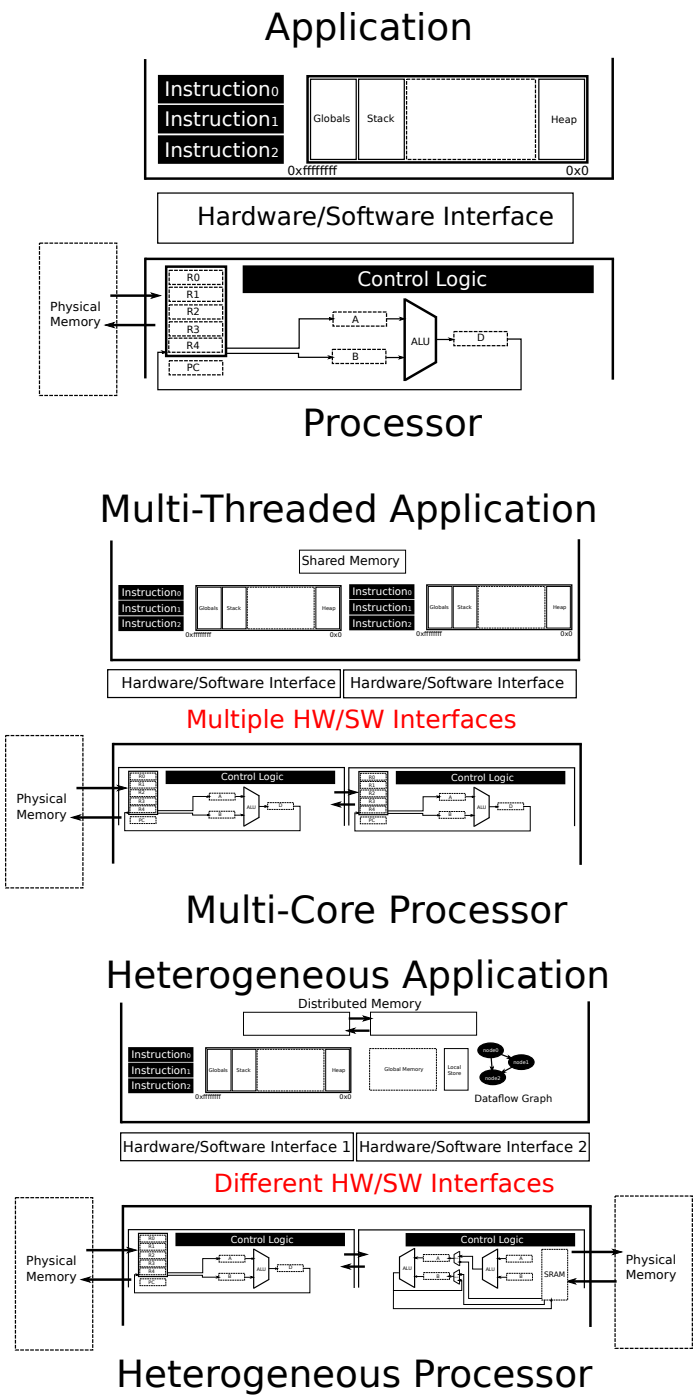

Figure 8: Multi-core and heterogeneous processors significantly increase the complexity of software design because they change the hardware/software interface. Multi-core processors require applications to explicitly coordinate among multiple instances of the same interface. Heterogeneous processors require applications to coordinate among multiple *different* interfaces.

CHAPTER IV

THE HARMONY EXECUTION MODEL

In the context of these prior efforts, this thesis introduces the Harmony execution model to address the gap between software complexity and hardware performance in heterogeneous systems. Rather than forcing the developer to manage multiple cores per processor and multiple processors with different micro-architectures and ISAs, Harmony applications are expressed in an Internal Representation (IR) that is mapped to processors in a heterogeneous system at runtime. The IR is intended to be the target of a high level compiler. It consists of a control and data flow graph of encapsulated data-parallel tasks, which are referred to as kernels, and managed resizable regions of memory, which are referred to as variables. Kernel-level-parallelism is used to distribute kernels among multiple processors and data-parallelism is used to map kernels to multi-core processors. The model is based on dynamic detection and tracking of data and control dependences(which are exposed in the IR), and a decoupling of kernel invocation and kernel scheduling/execution. The approach is inspired by solutions to instruction scheduling and management in OOO superscalar processors, where those solutions are now adapted to schedule kernels on diverse cores. Flow and control dependences are first inferred for a window of kernels that have yet to execute and then used as constraints to a scheduler that attempts to minimize execution time or power consumption while satisfying these dependences. A dynamic compiler is used to generate binaries for available processors and an online performance monitoring framework is used to collect information about kernels as they are executing and refine the kernel scheduler.

4.1 The Harmony Machine Model

In order to reason about the characteristics of applications that are expressed in an execution model, it is useful to define a model for the machine on which these applications will eventually run. The machine model assumed by Harmony is composed of a collection of processors and memories that are controlled by a central unit that coordinates their execution.

4.1.1 Processing Elements (Processors)

Processing elements are responsible for executing compute kernels. They may be restricted to execute only a static set of operations as in a fixed function ASIC or a processor functional unit. They may also be general purpose cores that are capable of executing arbitrary applications. In this case the same computation may be distributed across multiple processors, and there will be many possible schedules that assign kernels to processors, each with varying throughput, efficiency, and latency. A multi-core CPU could be modeled as a processor. A Graphics Processing Unit (GPU), a Digital Signal Processor (DSP), a fixed-function encryption engine, a processor functional unit, or a cluster of similarly configured nodes could be as well.

No assumptions are made about the performance characteristics of processors. Each processor in the system may perform differently, and that these characteristics may change over time and with applications and their inputs. For example, fault-tolerant processors may gracefully fail, reducing performance over time rather than failing catastrophically. Several existing systems perform voltage and frequency scaling in order to reduce their temperature or power dissipation in response to environmental events. Static models that assume a single metric of performance for each processor in the system may fail to capture these complex interactions, resulting in inaccurate and less useful performance predictions. The general approach used in this thesis is to perform profiling as the system is running to build performance models

over time.

Processors may support concurrent execution of multiple kernels. In this case, kernels may compete for internal processor resources, which may result in performance degradation of kernels that are executing together. Many general purpose processors and accelerators support the concurrent execution of multiple threads that share resources such as cache, registers, interconnect bandwidth, or a portion of a power-budget. Modeling these interactions rather than treating these processors as independent units allows a dynamic scheduler to choose between executing a kernel on a less efficient processor, or a highly loaded but efficient processor.

Processors are assumed to execute single kernels independently. Although it may be possible for multiple cores to coordinate together to execute a single kernel by decomposing it into distinct tasks that are allocated to individual cores, this model treats the aggregation of multiple cores as a single processor. A consequence of this decision is that there may be several possible groupings of cores into processors. This is particularly relevant for many-core accelerators such as Intel's Larrabee [130] or NVIDIA's Fermi [111] where multiple kernels can execute simultaneously, using a subset of the available cores. Rather than forcing the execution model to deal with the problem of changing the definition of a processor at runtime, all of the cores are treated as a single processor that can potentially execute multiple kernels simultaneously. In this case, internal scheduling mechanisms are used within the processor to map tasks to cores.

Processors that may be shared among several kernels are required to provide certain guarantees in terms of protection and isolation. Specifically, the following constraints must be enforced:

- **Memory Protection**: Memory spaces of the concurrently executing kernels must be kept distinct such that it is impossible for one kernel to manipulate (read or write) the scratch memory state of any other co-scheduled kernels.

- **Fault Isolation**: Catastrophic failure of a single kernel must not affect the execution of any other co-scheduled kernels. The processor must continue to execute any non-faulting kernels during any possible failure mode of another kernel. Failure or incorrect behavior of a kernel should not cause a processor to deadlock, physically break, or allow its memory to become corrupted. Additionally, individual failures must be strongly associated with a particular kernel. It is not enough to indicate that a fault occurred, the offending kernel must also be identified and the fault may be required to be deferred until some later point.

- **Forward Progress**: Kernel failures may manifest themselves as infinite loops. In these cases, forward progress of non-stalled kernels must be ensured until the system can eventually detect a mis-speculation as the cause for failure and kill the live-locked kernel. This may be accomplished using hardware multiplexing of resources or software schemes that periodically interrupt the execution of kernels and switch in others.

Collectively, these requirements enable the support of precise exceptions, speculative execution, and checkpoint/rollback fault tolerance. The Harmony execution model allows kernels to be executed out-of-order and scheduled along-side other independent kernels that occur previously in the sequential order of the program. If these out-of-order kernels generate faults, then the faults must not prevent earlier kernels from executing, otherwise memory may be left in an inconsistent state when the fault is reported and potentially handled. Similarly, out-of-order kernels should not gate the progress of other kernels to prevent previous kernels from never finishing, and possibly generating faults or terminating the program.

4.1.2 Memory Spaces

The second basic machine abstraction is that of a memory space. Memory spaces are visible at the system level, have fixed size, and are accessed via explicit connections

to processors or other memory spaces. Kernels that execute on processors that are directly connected to a memory space may access allocated regions directly, assuming that the kernel is given the appropriate permissions. Memory spaces support copying data from one space to another (including itself), allocating space for use by a kernel, and reclaiming allocated space via the following primitive operations:

- **Allocate** : Allocates a contiguous block of memory for use by a kernel in a directly connected processor. This operation fails if there is no space remaining. Allocation specifies the exact address within a memory space to allocate a variable.
- **Deallocate** : Frees an existing allocation.
- **Copy** : Copies a series of data from one allocation to another. The source and destination must be either allocations in the same memory space, or allocations in directly connected memory spaces.

Higher level operations such as reallocation (move/resize), compaction, or copy-on-write, can be implemented as a series of these primitives. Remote copies between loosely connected memory spaces may be accomplished via a series of point to point copies. Data streaming may be accomplished with a series of small copies together with double-buffering. Operations on independent allocations are allowed to proceed in parallel, whereas dependent operations are serialized. The runtime component of Harmony is allowed to reallocate and rename memory to eliminate false dependences and trade a larger working set for more memory-level-parallelism. This operation is only valid when using the primitive block copy operations rather than individual read-write operations within kernels that are not tracked by the runtime. It is also worthwhile to note that a series of allocation and deallocation operations may result in a fragmented memory space. It is the runtime's responsibility to manually compact

the memory space to improve performance or reclaim larger regions of contiguous memory.

All accesses to the same memory space are modeled as having the same access time. For example, a memory system consisting of multiple banks with addresses that are striped across banks is considered to be the same as a single memory bank. Relatively small distributed memory systems can be modeled by creating several individual memory spaces that are all connected to the same processor. However, in this case, memory allocations must be added to a single memory space only, and this practice only makes sense for kernels that operate on many allocations simultaneously. More structured models, such as Chapel data distributions [40], Sequoia trees [50], or Habanero Hierarchical Place Trees [154], may be better suited for a large number of memory banks, or a multi-level memory hierarchy.

The entire system of memory spaces and processors can be represented using a graph where vertices represent either processors or memory spaces, edges between processors and memory spaces indicate that kernels executing on the processor may access data stored in the memory space, edges between memory spaces indicate that copies between the spaces are allowed, and edges between processors are not allowed. Allocations that are accessible to a kernel running on a specific processor are referred to as local. Allocations that are required by a kernel, but located in a memory space that is not directly connected to the processor on which the kernel is executing are referred to as remote. All communication takes place via copies between memory spaces. The complete graph of memory spaces must be fully connected to ensure that arbitrary programs can be executed on the system. Note that some special cases of applications may include independent chains of kernels, in which cases all resources can be utilized with a less than fully connect graph. However, this special case is not considered because it does not support all possible applications.

Processors may include scratch space used for kernel computations that are not exposed in this model. These scratch spaces may be the on-board DRAM used by many GPUs, user-managed SRAM such as the local stores in the Cell processor [28], caches and registers used by most general purpose processors, functional unit reservation stations, or message passing queues used in on-chip networks. These memories are not persistent across a kernel's execution. In this model, they effectively become part of the processor that they are associated with as staging areas for data that is operated on by compute kernels.

There is no automatic memory coherence model. The runtime is responsible for making data stored in remote allocations available via explicit copies for local execution. The memory consistency model assumes that all writes from kernels to local allocations will complete before the termination of the kernel. This model allows only a single kernel to write to an allocation at a time. However, multiple independent kernels may read from the same allocation. Kernel schedules that result in undefined behavior under this consistency model, such as allowing multiple kernels to simultaneously write to the same allocation, are disallowed by the runtime.

4.1.3 Central Control

The execution of a Harmony application is orchestrated by a central control unit that is responsible for issuing commands that create allocations in available memory spaces, launch kernels on available processors, and enforce synchronization among these operations. Processors are treated as work units, while the central control unit is responsible for determining the next kernel to execute and dispatching it to an appropriate work unit. In this model, the central control unit may be a software runtime system executing on a single processor, or the fetch, decode, and issue stages in an OOO pipeline.

The use of a central point of control allows a runtime component to schedule a

dependency graph of kernels without the need to partition the graph and distribute it among multiple control units; this simplifies the runtime. However, it also limits the scalability of the model beyond a modest number of processors because the speed at which kernels can be dispatched is limited by the speed of the controller. Furthermore, central dispatch requires kernels to be sent over a network to the processing element on which it should execute. This is a relatively cheap operation with a few number of processors that quickly becomes latency intensive if the kernel launch command must be sent over many hops in a network. Although it is feasible that the functionality implemented by the central controller could be distributed over a series of components to improve scalability, this extension is outside of the scope of this dissertation.

The central control unit requires a mechanism for dispatching kernels on associated processing elements. Although this model does not specify the exact method of communication, it places several requirements on any implementation:

- Kernel Launch: A mechanism must exist for launching a specific kernel on a processing element in the system. The control unit must enforce in-order launch if it is required, and the launch mechanism may reorder kernels that are launched back to back without explicit synchronization.

- Launch Acknowledgement: A mechanism must exist for notifying the controller that a kernel has completed execution along with the status of its execution in the case that the execution produced an error.

- Kill Kernel (optional): A mechanism to halt the execution of a currently running kernel is needed to support certain types of errors that arise due to speculative kernel execution. This mechanism is not required if speculative execution is not employed.

These required mechanism may be implemented in multiple ways. For example, each processor could be running a server process that listens for messages from the

central controller. All processors in the system might be part of the shared address space where a threading programming model is used to synchronize communication through a set of shared variables between the central controller an the processors.

The central control unit is responsible for allocating memory, and scheduling kernels and memory operations. Depending on the capabilities of the central control unit, different methods for scheduling kernels may be employed. The primary constraint in all cases is that overheads of kernel scheduling must be small enough such that kernels can be dispatched fast enough to fully utilize the rest of the system. As the central control unit is made relatively more powerful, it can perform relatively more complex analyses to aid in the determination of a good kernel schedule. In the case where kernels are relatively coarse-grained and the central control unit is a complete general purpose processor, it may be possible to apply machine learning and statistical modeling techniques to predict the execution time of kernels and improve the quality of a schedule. In other cases where the control unit is a simple state machine, scheduling decisions may need to be made in a single cycle, and only simple checks for resolved dependences and available functional units may be possible.

One of the primary functions of the control unit is to fetch kernels from the program binary. Many OOO processors are bottlenecked by this operation; they are required to fetch multiple instructions each cycle, stressing the instruction memory hierarchy in cases when a series of instructions spans multiple cache lines or creates a cache miss. This is less of an issue for multi-accelerator systems because kernels are typically represent much larger operations and the control unit is much more capable. However, the ability of the control unit to scan ahead many kernels allows it to discover independent chains of kernels that can expose additional parallelism. As more processor resources are added to a system, the control processor must be able to fetch a larger window of kernels and employ more advanced techniques such as control decision prediction to accomplish this.

4.2 Model Abstractions

The core of the Harmony execution model is relatively simple. Programs are expressed in terms of compute kernels, control decisions, and variables, which are analogous to arithmetic instructions, branch instructions, and registers in most ISAs. Programs are specified as a sequence of kernels and control decisions, which can be expressed in a Control-Flow Graph (CFG) form, which will be covered in detail in Section 4.3.2.

4.2.1 Compute Kernels

Kernels are analogous to functions or procedure calls in imperative programming languages. However, this model places restrictions on compute kernels that enable fast inference of parallelism and independence from processor architecture. First, all input and output variables must be known when the kernel is invoked, i.e., no pointers embedded in the input/output arguments. This ensures that executing a kernel will not have any side-effects. Second, a kernel must use a single processor exclusively; it cannot, for example, distribute itself across a CPU and a GPU and manage GPU-CPU communication itself. However, it forbids global synchronization between groups of threads. Third, all input variables must be initialized before a kernel is executed. Finally, kernels may dynamically allocate local memory for scratchpad computations, but this memory may not be persistent across kernel executions. Persistent memory should be specified explicitly as a kernel output variable.

These restrictions give compute kernels the following properties:

- *No Side-Effects* : Kernels are only allowed to modify variables that are declared as outputs prior to the execution of the kernel. The execution of a kernel cannot modify any other state in the system.

- *Atomicity* : From the perspective of the Harmony runtime, kernels are atomic operations that require the existence of a set of input variables and change the

state of a set of output variables. The state of the output variables is considered inconsistent until a kernel finishes execution.

- *Exclusivity* : Kernels are executed exclusively by a single processor with a single memory space. They are able to use all of the resources of that processor without interference from other kernels.

- *Determinism* : Kernels are defined in a specific order subject to explicit control decisions. Although kernels may be executed out-of-order, any valid execution of a Harmony program must produce results that are identical to a sequential execution of the program. Coupled with the requirement that kernels may not read uninitialized data, this ensures that Harmony programs always execute deterministically as long as individual kernels are also deterministic.

4.2.2 Control Decisions

Control Decisions are syntactic structures for specifying control-dependent kernel execution and are analogous to branches in that they can potentially change the sequential flow of a Harmony application. Control decisions take in a set of input variables and determine the next compute kernel or control decision to be executed. Control decisions can be conditional or unconditional, and can include multiple possible targets. However, unlike indirect branches in most ISAs, control decisions must specify all possible targets explicitly. The explicit specification of such control dependences enables transparent optimizations such as speculation and predication to improve concurrency at the level of compute kernels in the same manner that branch prediction improves instruction level parallelism.

4.2.3 Variables

Variables in Harmony are managed explicitly to enable runtime support for variable renaming, migration across address spaces, and space optimizations. Variables

can be single elements, variably sized arrays, or complex data structures. However, in order to move variables between address spaces, complex data structures require opaque allocation, construction, and copy operators to be implemented by application developers or intelligent compilers. The ability to manage variables explicitly in the runtime allows for copy-on-write behavior to improve concurrency, imposing additional dependences on outstanding kernels to conserve memory, and performing static memory optimizations on the Harmony IR.

4.2.4 Discussion

Kernel Complexity There is no restriction on the type of operation that can be performed by a kernel or control decision. It can be a simple 32-bit addition or a complex sparse matrix factorization. The fact that kernels may perform different operations implies that they may have different complexities. This is true even of simple RISC instructions such as addition or logical operations, which may have different latencies on different functional units. Even in instruction sets, instructions may have data-dependent latencies. Divide instructions, for example, typically are implemented with state machines that iterate until a result is converged upon. This is even more relevant in the case of complex kernels, where latency may vary significantly from one kernel to the next, between different implementations of the same operations, and among different instances of the same kernel executing on different processing elements.

In/Out Kernel Parameters Generalizing the concept of input and output parameters from abstractions in instruction sets requires special consideration concerning output parameters. Most ISAs specify outputs as registers containing a single words of data. The treatment of data on a single-word granularity is significant. It implies that every time the register is updated, its current value is clobbered and overwritten by the result of the instruction. This behavior enables many useful optimizations

such as value prediction and register renaming. As kernels represent more complex operations, it becomes useful to specify operands as contiguous regions of memory rather than as single words. This creates semantic problems when a kernel only modifies a subset of an entire region. To handle this ambiguity, output parameters are either specified as pure outputs, or as input/outputs. Input/outputs must contain the previous contents of the variable, whereas pure outputs assume that the value has been clobbered, and any subset of the region that is not overwritten will be left undefined.

Kernel Aggregation Aggregation is a process by which multiple kernels from the same application are grouped together into a single larger kernel. In this case, the union of the kernel parameters that are not completely contained within the larger kernel (that is to say, those that either depend on an external kernel or those that an external kernel depend on) become the larger kernel's parameters. The variables that are completely contained within the kernel are allocated using scratch-pad storage. The larger kernel is a functionally equivalent operation with the same input/output behavior as the set of original kernels. In this way, it is possible to form arbitrarily complicated kernels from a series of smaller kernels. In fact, the high level kernels that are mapped onto accelerators can be thought of as an aggregated series of instructions, and a complete Harmony application can be thought of as an aggregation of kernels at the highest granularity available. This process will be used later in Chapter 5 to show how this model can be applied simultaneously at multiple granularities. However, as stated in the introduction, whether this model or another more structured model is employed at any particular granularity depends both on the structure of the application being expressed, and the organization of the system performing the computation.

Co-locating Kernels and Variables A consequence of the aspect of the machine model that allows processing elements access to only a subset of memory spaces in the system is that the full utilization of loosely connect processing elements requires the explicit copying of variables from one memory space to another during execution. This may introduce a significant performance overhead that outweighs the advantages gained from parallel execution or running a kernel on a more highly optimized accelerator. This is a common problem that arises due to the fact that it is very difficult to build a single centrally located memory system with high bandwidth and low latency. In other execution models, this problem is exposed directly to the programmer via DMA operations in CUDA and explicit message passing in MPI [117]. Although some models, such as those that rely on Partitioned Global Address Space (PGAS) [96] concepts, attempt to hide it via relatively inefficient remote memory accesses, the programmer still must be cognizant of the manner in which data is accessed to prevent an excessive amount of remote memory traffic. The approach advocated for the Harmony model is to delegate the problems of data partitioning, distribution, and migration to a runtime component. As with other aspects of the execution model, this strategy is most advantageous when the problem is more unstructured (e.g the data sizes change for different invocations of the same kernels) and the degree of parallelism is modest. The later sections of this dissertation explore runtime solutions to this problem in more detail that consider the dynamic amount of data that needs to be migrated, the load of remote and local processing elements, and the expected speedup from parallel or accelerated execution.

4.3 Model Analysis

Once an application has been expressed using the Harmony execution model, it is often desirable to reason about its behavior or perform transformations on it that affect its performance, rather than its correctness. This section describes several

common analyses that provide information about constraints on potential scheduling orders, enable the expression of an entire application as a graph of kernels supporting various transformations, and metrics that can be used to reason about the relative compute complexity of kernels. Kernel data and control dependences place fundamental restrictions on the scheduling order of kernels and the parallelism in a Harmony application. Control flow and data flow graph representations can be used to capture these dependences in a form that enables optimizations to be applied to the graph.

4.3.1 Kernel dependences

Kernel dependences are inferred automatically via a combination of the sequential ordering of kernels, which defines a consistent view of memory after each kernel completes, and the explicit specification of kernel inputs and outputs, which denote the changes that the execution of the kernel will make on the memory state. Data and control dependences place constraints on the execution order of kernels such that this consistent view of all variables in the system is maintained. Data dependences prevent kernels from reading values that have yet to be generated, and over-writing results that are needed by another kernel. Control dependences indicate a relationship between the generation of data and the next set of kernels to be executed. They state that the result generated by a kernel will determine the next kernel to execute.

Flow dependences Flow dependences are also known as read-after-write or true dependences because they indicate that the results of one kernel are fed into another kernel. Flow dependences are fundamental limitations on parallelism because they specify a producer-consumer relationship between kernels. Although some techniques such as value-prediction [22] and lazy evaluation can be used to execute kernels with flow dependences in parallel, the success of these techniques depends significantly on the characteristics of the application. Finally, it is worthwhile to note that a flow dependency between kernels does not necessarily mean that changing the order

of their execution will result in an incorrect operation. Some operations, such as reductions, can be performed as a series of operations with flow dependences or as a tree of semi-parallel operations with flow dependences between successive levels in the tree rather than between each successive kernel. Other operations may ignore inputs in specific circumstances, resulting in a conditional flow dependency. However, the both of these cases require domain specific knowledge of the behavior of kernels, which are abstracted as opaque operations in this model.

Anti dependences Anti dependences, or write-after-read dependences, exist to prevent a kernel from writing over the results of a previous kernel before it is completely read by a third kernel. These dependences occur when the same variables are multiplexed among several kernels to reduce the memory footprint of an application. If a result is generated and then immediately consumed by another kernel, the associated storage can be later re-used by other kernels. When these kernels are executed in parallel, multiple copies of the variable are needed to allow both sets of kernels to access them at the same time. There is a trade-off between allocating additional storage to support parallel execution and that will be explored in more detail in the optimizations in Chapter 6 and the case studies in Chapter 8.

Output dependences Output dependences are sometimes referred to as write-after-write dependences. They are meant to prevent variables from being updated in the wrong order. So that the final a variable that is written to by a sequence of multiple kernels will be the value set by the last kernel, even if the kernels are executed out of order. These are similar to anti dependences in that they allow multiple sets of kernels to share the same variables for communication. They may also occur as the result of dead code that produces a value that is never consumed before immediately being written over.

Relaxing Data dependences In some cases it is desirable to relax the requirement that kernels execute atomically in order to further optimize the execution order of a series of kernels. This is only useful in cases where kernels will complete a read or write operation on some parameters before others. The most obvious case is that many instructions will first read all input operands in the early stages of the pipeline, and only commit the results immediately before the instruction is retired. In these cases it is permissible to start other instructions that write over the inputs before the dependent instruction has been committed as long as the inputs have been read. Generalizing this concept to higher level kernels would require a mechanism for indicating that input/output operations on kernel parameters have completed. This optimization is not included in the model or examples presented in this dissertation. However, it is mentioned here for completion as the additions to the model required to support it would be relatively lightweight.

Control dependences Control dependences indicate that the execution of a series of kernels is predicated on the sequential flow of control through the programming passing through them. Typically control dependences originate from a control decision and end at all kernels that are in the post-dominance frontier of that control decision. This indicates that those kernels will only be executed if that control decision does not skip them. Control decisions may depend on other control decisions, and although it may be necessary to pass through multiple control decisions from the program entry point to reach a kernel, only the immediate dominator, rather than all dominators, are listed as the sources of control dependences. This is a convention that is adopted in this work for simplicity in the same way that direct data dependency chains are specified rather than all indirect producer-consumer relationships. It is possible to reconstruct the set of all control decisions that a kernel depends on by rewinding through the graph of control dependences.

Kernel Level Parallelism Control and data dependences place constraints on the execution order of kernels. Whereas an application with no data dependences allows all kernels to be executed in parallelism, most applications allow only subsets of kernels to be executed in parallel. In order to distinguish between these types of applications it is useful to have a metric that quantifies the amount of parallelism in a given application. ILP is a metric that is commonly used when dealing with parallelism at the level of instructions. It is informally defined as the total number of dynamic instructions in an application divided by the longest path through the control and data dependency graph of those instructions. This is usually sufficient when dealing with instructions, and the different instruction latencies are typically ignored in literature. However, this simplification is harder to justify when dealing with higher level kernels; a single kernel may dominate the execution time of an application with many other independent kernels. To handle cases like this, the concept of ILP is augmented by weighting each kernel by its execution time. This metric is used throughout the experimental sections of this dissertation.

4.3.2 The Kernel Control Flow Graph

The kernel CFG is a low level internal representation of Harmony programs. It is a directed cyclic graph where nodes are a series of interleaved kernel calls terminated by a control decision. Edges originate at control decisions and end at possible targets of a given control decision. Only the first kernel in each node may be the target of a control decision and each node may have at most one control decision. This ensures that nodes have a single entry point and a single exit point. In this representation, the common term Basic Block (BB) is used to refer to a node. The only difference between this notion of a BB and the classical notion is that these BBs contain kernel calls rather than instructions. Note that this is not the same as a control dependence graph.

4.3.3 Dependency Graph

It is also useful to have information about the data dependences of kernels in a program. The kernel DFG augments the CFG with a global notion of data-flow through the program. It is defined both within a BB and among basic blocks. Within a block, nodes represent kernels and edges represent data dependences. Edges originate at kernels and end at dependent kernels. Moving up to the level of basic blocks, it is possible for kernels to be data dependent on previous kernels and control dependent on previous control decisions.

Dynamic Data Dependency Graph The dependency and control flow graphs express static information about a Harmony program. It is also useful to incorporate runtime information about the actual behavior of the program into this analysis in order to inform decisions that are made dynamically. A dynamic data dependency graph does just this. It assumes that a window of kernels have been selected for execution, but not yet scheduled on processing elements. dependences are tracked only for these kernels. New kernels that are fetched from the program and selected for execution are added to the dependency graph. Those that are present in the graph with no dependences are eligible for scheduling. Those that complete execution are removed from the dependency graph, removing dependences from the graph and potentially freeing existing kernels for execution. This data structure is commonly used in processor scoreboards.

4.3.4 Performance Analysis

Along with static information concerning program data and control flow, it is also useful to consider the performance characteristics of kernels and their influence on the complete application. This is necessary to perform scheduling optimizations predictions of an application's behavior. Kernel complexity attempts to determine the

relative amount work performed by individual kernels. Architecture efficiency captures the relative performance of the same kernel on different processing elements. Criticality identifies kernels that are on or likely to be on the critical path through the application, creating bottlenecks on highly parallel systems. Memory intensity combined with working set size provides information about the amount of memory accessible by individual kernels, and how frequently this memory is actually accessed.

Kernel Complexity As the level of abstraction of a kernel is raised to higher granularities, the computations that are performed by kernels become increasingly complex. In most cases, the complexity of a kernel is a function of its input parameters, the calculation that it performs, and the processing element upon which it runs. In order to schedule kernels effectively, it is necessary to have accurate a-priori information about the expected execution time of individual kernels. This problem of modeling kernel execution time becomes increasingly difficult as kernels are allowed to perform more diverse classes of operations. Additionally, even kernels that are deterministic in the sense that they always produce the outputs for a given set of inputs may have non-deterministic performance characteristics. Resource sharing, voltage and frequency scaling, or the use of random algorithms that eventually converge on a deterministic solution may cause the complexity of the same kernel running on the same processing element with the same input parameters to have different execution times.

Architecture Efficiency The use of accelerators as processing elements introduces an aspect of kernel scheduling concerned with mapping kernels to efficient processors, rather than arbitrary processors. This scheduling process is facilitated by information regarding the relative efficiency of kernels on different types of accelerators. This model makes no assumptions about the types of processing elements available in a system or the types of operations implemented by kernels. Kernels that either make

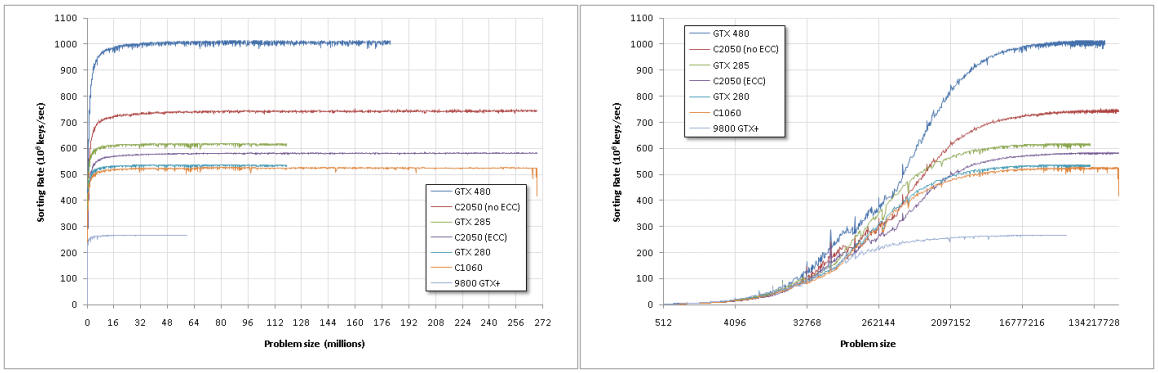

Figure 9: Scaling of a GPU radix sort algorithm from Merril et al [104].

use of dynamic compilers or include multiple implementations may perform more or less efficiently depending on the processing element that is used to execute them. Although it would be possible to consider different metrics for efficiency, such as wall clock time, hardware cycles, total energy consumed, power, or various other metrics, this dissertation considers wall clock time as a proxy for performance. Wall-clock time was chosen because it is relatively easy to measure, and because is not biased by tuning operations like frequency and voltage scaling.

Efficiency is a function of the kernel input parameters, the architecture of the processing element performing the execution, and the actual computation performed by the kernel. However, in many cases, the *relative* efficiency may only be a function of the architecture and the kernel, making it possible to determine an efficiency metric that can be computed statically and remain useful at runtime. For example, Figure 9 shows the performance of the same sorting kernel on six different hardware configurations, and that the relative performance remains fairly constant over a range of data inputs. Creating a more general metric requires runtime, profiling, or analytically derived information about the dependence on the inputs supplied to the kernel. A case study in Chapter 8 addresses this more general problem.

Criticality The concept of criticality comes from the observation that some kernels have significantly greater latency than others. In processors, these are usually

memory operations that trigger cache misses, whereas in general they are more difficult to predict. Identifying these kernels and beginning their execution as soon as possible has the potential to significantly improve processing element utilization and system throughput. It can also be the case that a critical kernel's execution is gated by dependences on other lower latency kernels. Systems that attempt to schedule kernels by criticality typically consider all kernels in dependency chains ending in high latency kernels as being critical as well. et al. [140] show that it is possible to predict criticality based on the number of dependences originating from a kernel as well as the kernel mix. Intuitively, kernels with a large number of dependences have a greater probability of restricting the execution of later, high latency high complexity kernels.

Memory Intensity Data that flows between kernels is captured in kernel parameters and the memory spaces that they are assigned to. However, kernel parameters only indicate that a kernel may need access to the data stored in a variable. Recall that variables may be single elements or simple linear arrays, but that they may also be complex data structures. In these cases, it is possible for a kernel to access the complete data structure or only a few elements that it contains. These access patterns are captured in the memory intensity of a kernel, which describes the expected amount of memory traffic between the kernel and individual variables. This information is useful when scheduling kernels in cases when memory spaces in the system have different performance characteristics, as well as taking care not to oversubscribe the available bandwidth of any memory space.

Working Set Size Memory intensity is related to the working set size of a kernel. Whereas memory intensity measures the amount of memory traffic between a kernel and its parameters, working set size measures the amount of data in each variable that is actually accessed by a kernel. Processing elements that support hardware

caching or zero-copy memory management such as NVIDIA Fermi [110] may benefit from execution of kernels with low memory intensity, but large working set sizes. Additionally, information about the expected working set size of a kernel can be used to match a working set size to scratch spaces or local memories in processing elements. Working set information is typically difficult to predict. However, it is possible to measure it dynamically as a kernel is running, and potentially re-schedule the kernel or bias its subsequent schedule based on this information.

4.3.5 Reorganizing an Algorithm

As a final comment, the reader should recognize that there is a limit to the types of analyses and the types of transformations that can be performed on this program representation. It is relatively easy to perform optimizations on the level of the dependency graph structure, but it is relatively difficult to recognize the algorithm or data structures that are used to execute an application. Additionally, the explicit partitioning of an application into kernels and the treatment of kernels by this model as opaque, atomic operations makes it very difficult to discover domain specific information about the behavior of a kernel. For example, it would be very difficult to determine that a particular kernel was performing a sort operation rather than a generic permutation without observing its internal implementation.

4.4 Expressing Applications in Harmony

4.4.1 High Level Languages

This section discusses several high level languages and describes how each of them could be transformed into the Harmony model with varying difficulty. A high level observation is that any language may be transformed into the Harmony model as it is Turing Complete. However, languages without explicit notions of compute kernels either require compilation down to the lowest level of encapsulated operations (usually instructions), or the compiler to form kernels automatically via aggregation or other

algorithmic transformations that maintain the function of the program while changing the structure of its dependences.

StreamIt This language expresses computations as a cyclic graph of compute kernels. Compute kernels in StreamIt share many of the same properties as Harmony kernels, such as the side-effect free property of kernels. However, there are a few subtle differences, mainly due to the execution of StreamIt kernels being triggered by the arrival of new data on the kernel's inputs. This is significantly different from the kernel launch mechanism in Harmony, which includes a sequential thread of control to drive kernel launch and execution. However, it is still relatively simple to express StreamIt applications in Harmony by partitioning the StreamIt data-flow graph into acyclic sections. In each section, the kernels can be scheduled topologically and include a predicate variable for each kernel output. All dependent kernels can be guarded with a control decision that gates their execution depending on the value of the associated predicate. This allows the graph to be collapsed into a sub-graph of acyclic nodes. The remaining cyclic sections can be transformed by setting another predicate for data elements that trigger the next kernel in the cycle. In this case, rather than predicating the execution of that kernel, a conditional backwards control decision may be used to jump back to that kernel if the condition is set, resulting in the potential re-execution of the entire chain of data and control dependent kernels. Using this process it would be relatively simple to build a compiler that would transform arbitrary StreamIt programs to the Harmony execution model.

CUDA and OpenCL CUDA and OpenCL are new languages that partition a complete applications into a mix between a C/C++ host component that periodically launches compute kernels, which are expressed in a Single-Program Multiple-Data (SPMD) form using a C-like syntax. The fact that these applications are expressed in terms of compute kernels precludes the need for the compiler to automatically discover

kernels. However, OpenCL kernels do not distinguish between input and output parameters, and CUDA allows arbitrary pointers to be used in kernel parameters. This problem can be solved completely for OpenCL by conservatively assuming that each parameter is input/output and then checking for operations within the kernel that modify or access the parameter to eliminate one of these conditions. CUDA applications are generally more difficult, requiring either complex pointer analysis to identify kernel parameters or very conservative assumptions, for example, that a hard to analyze kernel is dependent on all preceding kernels. The host code that is used by these languages to launch kernels is another source of difficulty, which is often easier to compile directly into individual instructions than requiring the compiler to automatically identify kernels. It is useful to note that several small additions to these languages, such as requiring the programmer to tag kernel parameters explicitly, would significantly reduce the burden on a compiler.

Haskell Haskell is a functional language with strong static typing. Functions in Haskell satisfy all of the properties required for Harmony kernels in that they are explicitly demarcated into inputs and outputs, and all side-effects are specified explicitly as function parameters called *monads*. This makes mapping from Haskell functions to Harmony kernels simple. Prior work has shown that it is possible to efficiently compile data-parallel operations in Haskell, such as map, reduce, or list comprehensions, into CUDA/OpenCL kernels [93]. Although it is outside of the scope of this work, these two processes could be combined by a compiler to convert native Haskell programs into the Harmony execution model where each kernel would have a CUDA implementation suitable for execution on a system composed of data-parallel accelerators.

C, Java, Fortran, and Others Traditional imperative languages such as C/C++, Java, Fortran, and many others express applications as a series of low level operations

that are grouped hierarchically into higher level procedures and data structures to express complex applications. Although the notion of a procedure call containing a group of related low level operations would seem to make an idea abstraction to translate onto the Harmony notion of a kernel, these languages typically allow procedures to have arbitrary side effects, access arbitrary regions of memory that are difficult in general to bound at compile time or even at runtime, and perform complex low-level interactions with the system such as changing their own instructions or directly interacting with hardware state that may not be universally available. These characteristics make it extremely difficult to map functions in these languages into kernels automatically. There has been some success mapping well defined structures in these languages into CUDA and OpenCL kernels for use on accelerators, for example, PGI-Accelerators [58, 59]. However, these efforts usually place many of the same restrictions on code that can be mapped into kernels as this model.

Array Languages - Enjing, Copperhead, Matlab Enjing [13] takes the approach of recognizing common numerical functions from a library of python extensions and converting them into kernels. Copperhead [24] goes a step further by mapping more generic data-parallel operations such as map, scan, and reduce over user defined data structures into CUDA kernels. Jacket extensions to Matlab [124] identify common functions used in Matlab and replace them with equivalent CUDA kernels. All of these approaches attempt to identify data-parallel array operations over recognized data structures.

Aggregate operations over regular data types, such as transformations over matrices, make good candidates for automatic promotion into kernels. Compiling languages with these features into the Harmony execution model requires identifying aggregate operations and the variables that they transform, and mapping these into equivalent Harmony kernels and variables. Other language statements that are ancillary

to the kernels, which do the heavily lifting in the application, may be mapped directly into lower level kernels. These will eventually be scheduled alongside the larger data-parallel kernels. As long as the overheads for tracking these smaller kernels is relatively small, these applications can be efficiently mapped to Harmony.

SQL, Datalog, First Order Logic Database programming languages are mainly derived from primitive operations common to first order logic. They are also declarative, meaning that they specify the expected result of the computation rather than a list of steps required to determine it. Due to their roots in first order logic, many database programming languages such as SQL and Datalog can be mostly or completely represented using Relational Algebra (RA) [31]. Relational algebra itself consists of a small number of fundamental operations, such as join, intersection, union, projection, etc. These fundamental operators are themselves complex applications that are composed of multi-level algorithms and complex data structures. Given kernel-granularity implementations of these operations it is possible to compile many high level database applications into the Harmony execution model by expressing them as a CFG of RA kernels. An example of such a compiler was implemented as part of this dissertation and is explored in a case study in Chaper 8.

4.5 Examples

The execution of an application implementing the Harmony execution model may be viewed as logically equivalent to that of a sequential application executing on a modern super-scalar, out-of-order (OOO), processor with instructions replaced by kernels and functional units replaced by heterogeneous processors. During execution, the runtime walks the CFG in program order. It also scans a window of kernels that have yet to be executed using either branch prediction or predication. Dependence-driven scheduling over kernels in a *dispatch window* deploys kernels on available processors.

```
1   int* p_hmaxs = h_maxs; // initial input data
2   float* p_hvars = h_vars; // initial input data
3
4     // Launch the device computation threads!
5   for (i = 0; i<t-block_num; i+=block_num)
6   {
7       // Launches a kernel
8       PetrinetKernel<<< grid, threads>>>(g_places, g_vars, g_maxs, N,
9           s, 5489*(i+1));
10      // Copies results from the Device to the Host
11      CopyFromDeviceMemory(p_hmaxs, g_maxs, block_num*sizeof(int));
12      CopyFromDeviceMemory(p_hvars, g_vars, block_num*sizeof(float));
13      // Selects the next inputs
14      p_hmaxs += block_num;
15      p_hvars += block_num;
16  }
```

Figure 10: CUDA source code for the inner loop of PNS. CUDA requires applications to be manually partitioned into GPU kernels and C++ statements that are executed sequentially. Harmony treats all kernels, C++ statements, and memory transfer operations as encapsulated kernels that execute out of order subject to data and control dependences.

Optimizations such as speculative kernel execution[1] are supported via the separation of kernel execution (affecting local state) and commitment of the results of execution (affecting global state).

4.5.1 Petrinet Simulation

The following example is used to illustrate the advantages and potential of this model of kernel level parallelism, as well as identify opportunities and challenges. It focuses on an example of a sample execution of two iterations through PNS, a petrinet simulation. This application was originally implemented in CUDA as part of the Parboil benchmark suite [72]. In this example, CUDA kernels, memory copy operations, and host code sections are mapped to Harmony kernels by hand to demonstrate how the process could be performed by a high level compiler. The high level structure of the application is visible in the original CUDA source code shown in Figure 10. In CUDA, the program loops over a range of input data and makes blocking kernel calls

[1] Speculative kernel execution is explored in detail in prior Harmony work [46].

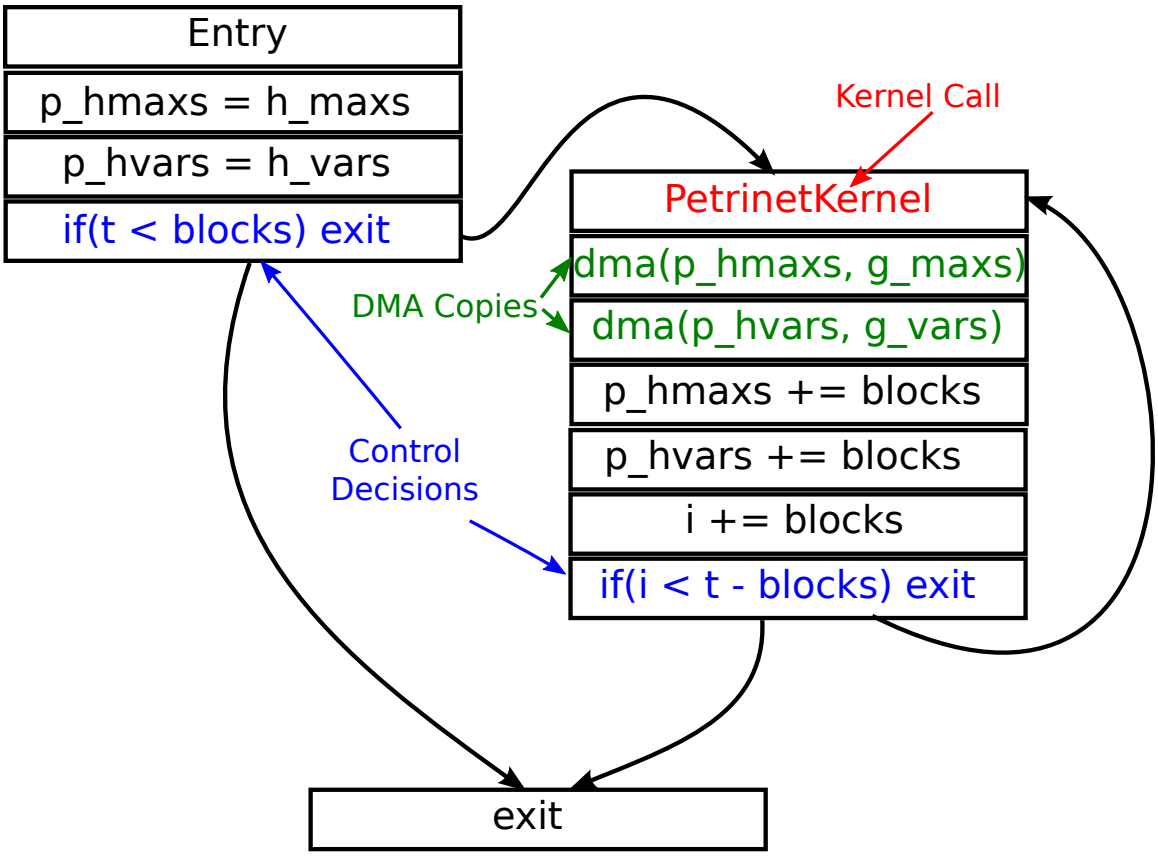

Figure 11: The control flow graph for the PNS application. This is the representation that Harmony programs are compiled into before being executed.

for each set of data. CUDA interleaves calls to compute kernels, which are dispatched to a selected GPU, with C++ statements, which are executed directly on a CPU. In Harmony, both the CUDA kernels and the C++ statements are mapped to Harmony kernels, control structures (the for-loop in this example) are mapped to control decisions, and regions allocated with cudaMalloc are mapped to Harmony variables.

Using Harmony, the PNS source code would be compiled into an intermediate representation that expressed the application only in terms of kernels, control decisions, and variables. Figure 11 shows the basic building blocks of the application represented as a control flow graph of kernels. Control dependences are expressed explicitly in this representation, which eases their resolution at runtime. PNS is typically used to model distributed systems. This particular version is composed of a

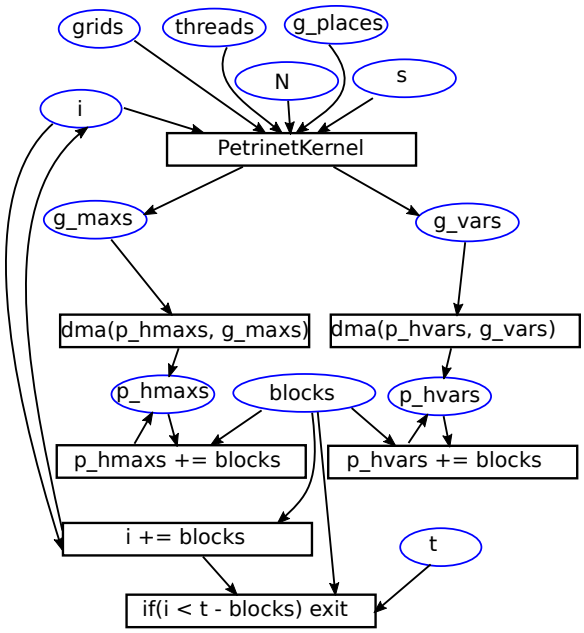

Figure 12: The data flow graph for one iteration through the PNS loop. The black nodes represent kernels and the blue nodes represent variables that are managed by the runtime. Edges represent data dependences.

loop around a single kernel and memory copy that updates global state.

When the Harmony runtime begins executing this program, it scans through the kernels as they are encountered in the sequential flow of the application without blocking. This process implicitly creates a window of kernels that have been encountered, but not yet executed. As the window is created, the runtime builds a graph representing all of the data dependences among kernels in the window. The data dependency graph after one iteration through the loop is shown in Figure 12. Control decisions limit the number of kernels that can be scanned without blocking because they potentially change the flow of the application. This limitation can be avoided using techniques like prediction or predication to speculatively scan kernels beyond a control decision as long as there is a method for recovering from miss-speculations. In Figure 13, the kernels from the second iteration have been invoked speculatively.

Another issue arise when two kernels share the same memory, but do not have a producer/consumer relationship. In Figure 11, all of the temporary variables are

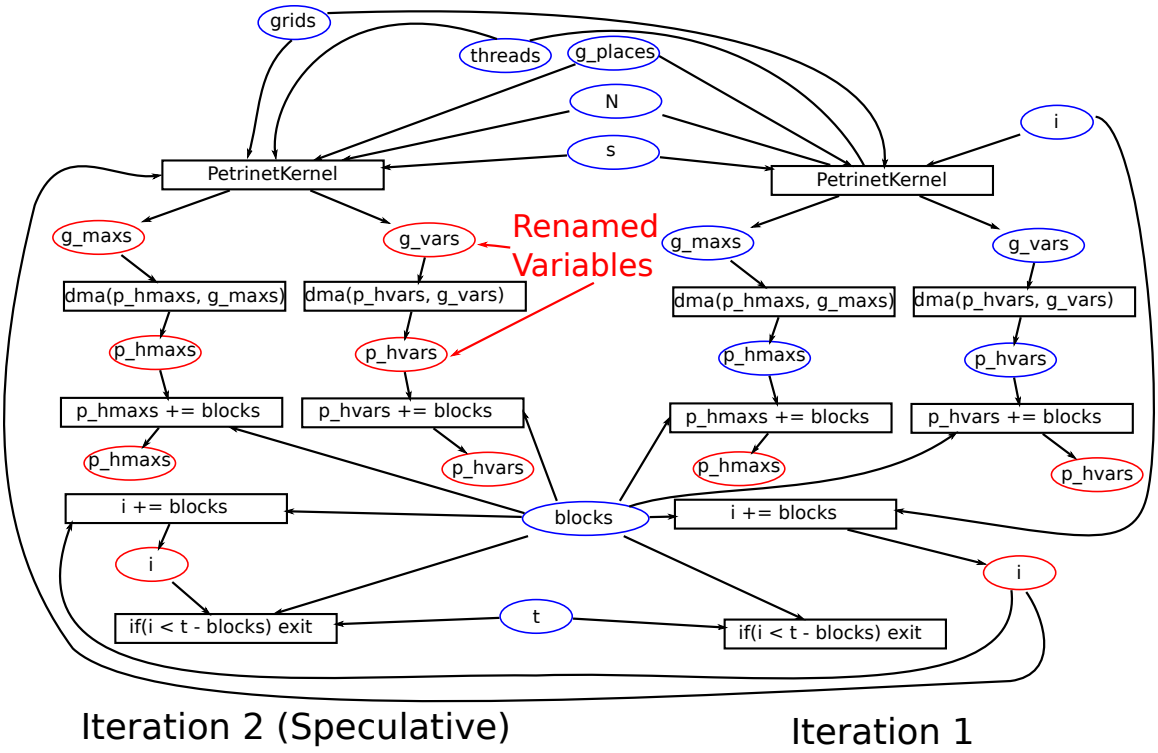

Figure 13: The data flow graph for two iteration through the PNS loop. Kernels from the second iteration have been invoked speculatively and red variables have been renamed to eliminate false data dependences.

reused through multiple iterations of the loop. However, the petrinet kernels do not have explicit flow dependences. These are directly analogous to output and anti dependences in instruction scheduling. Consequently, variable renaming is used to eliminate non-flow dependences: every time a non flow dependency is found between two kernels, the runtime allocates a new variable to be used instead, effectively breaking the dependency at the cost of increasing the memory footprint of the application. In the dependency graph shown in Figure 13, all of the variables denoted as red nodes have been renamed and replicated.

The runtime utilizes a machine model description (e.g., number and type of cores) paired with a dynamic compiler (Ocelot) to generate implementations for each kernel for at least one (but possibly many) ISAs. As kernels complete execution, dependences between kernels in the dispatch window are updated and the schedule is revised.

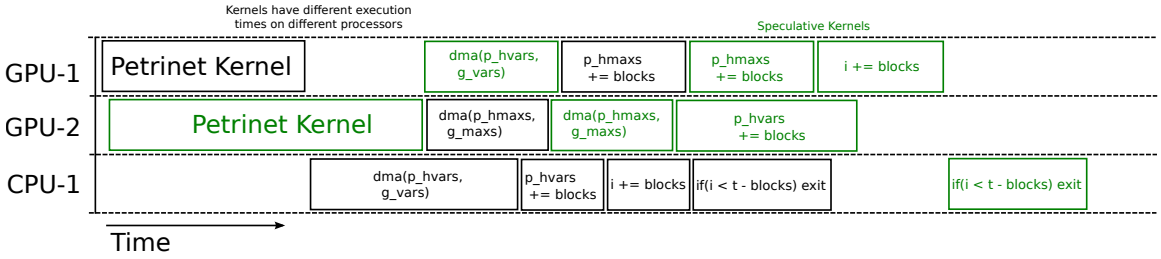

Figure 14: A possible schedule for two iterations of the PNS application on a system with two different GPUs and a CPU. The green kernels in the figure are from the second, speculative, iteration. Note that speculation significantly increases the parallelism in the application. It is also worthwhile to note that the same kernel can have different execution times on different processors.

Figure 14 shows an example schedule for the PNS application using a system with a CPU, and two GPUs with different micro-architectures. This scheduling phase creates a dynamic mapping from kernels to heterogeneous architectures, while the existence of binaries for multiple machines permits execution on systems with radically different processor configurations, and performance to scale as more resources are added to a system until kernel level parallelism is exhausted.

Note that the execution time of individual kernels varies across architectures and can even be data dependent (in Figure 14, the petrinet kernel runs slower on the second GPU), yet being able to accurately predict this time a-priori improves the quality of the schedule as pointed out in the first Harmony publications [47] as well as several newer studies performed independently [77,95]. To address this issue, online monitoring is used for measuring the execution time of each kernel as well as a set of kernel characteristics. It is then possible to construct kernel and input dependent models of execution time as a function of target core and utilize these to make more effective scheduling decisions. Work on this topic in the context of the Harmony execution model has been explored in studies on on workload characterization [84] and performance modeling [85], which are uniquely leveraged to build these models.

CHAPTER V

APPLICATION OF THE MODEL AT MULTIPLE GRANULARITIES

Although this dissertation focuses on the application of the Harmony execution model to systems with multiple accelerator boards, the machine model can be applied to any system that can be represented as a collection of processing elements and memories with different capabilities and performance characteristics. This section illustrates this point by showing how the individual abstractions of kernels, variables, and control decisions can be mapped onto functional units and registers in superscalar processors, cores on a System on Chip (SOC) processor, accelerator boards and DRAM in single nodes, and collections of similar machines in clusters or distributed systems. The expectation is that the right place to apply this model and the associated optimizations presented in the next section will shift as systems and programming models continue to evolve. It is also possible that this model could be applied simultaneously at multiple granularities in the same complete system.

5.1 Functional Units in a Superscalar Processor

All of these topics are covered in related work. They are presented here in terms of abstractions in the Harmony execution model with the intent of exposing the reader to them in a familiar form that can easily be related to similar concepts at higher granularities.

5.1.1 Pipeline Organization

This form of an OOO superscalar processor is common to commercial architectures such as the Intel P6 [120] and its derivatives [53] as well as competing architectures

from AMD [83] and ARM [36]. The pipeline is typically organized into stages that are connected by First-in First-out (FIFO) queues. Instructions are first fetched and decoded. The decoded form of each instruction is placed into an issue queue. Registers are renamed to eliminate false data dependences and issued from the queue as their dependences are resolved. Issued instructions are directed to functional units based on the type of operation that they perform, and are typically scheduled according to some metric such as age or criticality. Instructions that complete are held along with their results in a final buffer the Reorder Buffer (ROB) that prevents them from updating the register file or global state out-of-order. Results are committed from the ROB in order according to available register file bandwidth, and as instructions become non-speculative. Speculative instructions that occur due to predicted fetches or scheduling assumptions are tracked throughout the pipeline and discarded or replayed from the ROB if the speculation is determined to be incorrect.

Program Binary Format This class of processors execute applications that are stored as encoded binaries of instructions in memory. The encoding is typically done as a space optimization to reduce the bandwidth and space requirements of an application; instructions need to be decoded prior to execution. In terms of the Harmony model, these applications are stored as a compressed kernel CFG that is interpreted by the processor hardware.

Although it is generally possible to recover the complete kernel CFG from this form, most systems forgo analysis of this representation and instead choose to deal with a limited window of the complete application at a time. Note that some systems that are tightly integrated with dynamic compilers, such as Transmeta's processors [39] or Hydra [114], do more detailed analysis of compiled programs during execution by building a partial or full program CFG.

Instruction Fetch Instructions are light-weight enough that the cost of fetching them from memory is similar to the cost of executing them [63]. This characteristic results in several optimizations being applied to the simple process of reading the next instruction to execute. An instruction cache provides low latency and access to previously fetched instructions; it also fetches blocks of instructions that are likely to be executed along with the current instruction. Branch predictors are employed to predict the next set of instructions that are likely to be executed, and trace caches store them in a format that makes accesses to commonly fetched chains of instructions efficient. Instructions are encoded in a way that compresses their memory footprint, for example, the typical x86 instruction expands to over 70 bits of microcode when decoded.

It should become apparent that as kernels become more complicated, it becomes less advantageous to optimize the fetch stage.

Variables Are Registers Registers that are exposed in the ISA correspond to variables in the Harmony model. As instructions are readied for execution, they are mapped to physical registers through a renaming process. Physical register files are typically larger than the set of ISA registers to allow for many instructions to be inflight at the same time, each of them writing to a different physical register. Some systems such as the P6 allocate physical registers along with the reorder buffer entries for each instruction, simplifying allocation and deallocation. Other systems allocate physical registers out of a pool of pointers into the physical register file that is updated as instructions retire.

Memory addresses are typically not managed in the same way due to the comparatively large capacity of memory and the difficulties associated with tracking all updates to all bytes of memory. Many processors rely on a load-store buffer to track the last few updates to memory and to forward data from previous writes to reads

from the same location.

The Issue Queue The issue queue maintains a window of decoded instructions that are eligible for issue. As instructions are inserted into the queue, they are allocated physical registers, reservation stations, and other hardware state. Although many processors divide the issue queue into clusters [118] to reduce circuit complexity, they collectively perform the task of tracking dependences for a window of instructions and determining when they are scheduled on hardware. Multiple instructions need to be scheduled each cycle, limiting the complexity of any scheduling logic. Still, this can be accomplished via the broadcast scheme used in Tomasulo's algorithm [146] or other heuristics based on age [78] or criticality [140].

Functional Units Functional unit represent the processing elements in OOO processors. They are typically grouped together into units that can perform a set of related functions. They may support bitwise operations, integer addition/multiplication, floating point operations, vector operations, and load/store instructions. These are heterogeneous processing elements, not all functional units can execute all instructions. There is typically an instruction steering component of the instruction scheduler that is responsible for directing decoded instructions to only those functional units that are capable of executing. The latencies of many operations are known, and some schedulers use this information to schedule chains of instructions. However, once an instruction is scheduled to be executed on a function unit, the operation proceeds atomically. Some functional units are pipelined and allow multiple operations to execute concurrently, but typically only a single operation may begin execution each cycle.

Speculative State Instructions can be speculative due to incorrectly predicted branches, precise exceptions, or instructions that were scheduled assuming that some

non-deterministic event occurs, for example, that a load hits in the cache. When a mis-speculation occurs, the amount of state that has been modified incorrectly consists of physical registers that have been modified and load/store operations that have been scheduled. This state needs to be discarded before correct execution can resume.

Summary Collectively, the hardware that implements these functions represents the runtime component of Harmony, and the set of optimizations that are applied represent dynamic optimizations that are supported by the Harmony model. The optimizations that are used here are a subset of valid transformations on the model; the hardware cost of these optimizations is considered during design, and only those optimizations where the cost can be justified by performance improvements make their way into real designs. Although all of these optimizations are supported by the Harmony model, the trade-offs change when it is applied at different granularities.

5.1.2 Overheads

The relatively low complexity of instructions limits the amount of hardware that can be devoted to optimizations such as branch predictors, instruction schedulers, and caches. Instructions typically complete in one or a few clock cycles. The following discussion identifies those components that tend to consume the most resources.

Dependency Checking Another high overhead operation involves instruction selection and wake-up. When an instruction completes execution, it is necessary to notify dependent instructions so that they may become eligible for scheduling. The original form of Tomasulo's algorithm required a tag associated with each operand to be broadcast to all waiting instructions and compared with the tags of their source operands. In this case the waiting instructions are in functional unit reservation

stations or the issue queue. As these queues increase in size, the wires from the functional units become longer, and the number of comparators required to perform the broadcast and tag-check increases.

Instruction Fetch The coded form of most instructions further complicates the fetch process by making it difficult to determine where the next instruction begins before fetching and decoding the current instruction, especially in the case of a branch. This typically results in a long critical path through decoding logic, which also needs to handle the case where a series of instructions span multiple cache lines, and are aligned to different words in the current cache line. As more instructions are decoded in parallel, it becomes necessary to fetch multiple cache lines per cycle, shuffle their bytes around to feed into the appropriate decoders, and identify the boundaries between instructions.

Operand Forwarding Operand forwarding imposes an overhead associated with moving the results of an instruction to another functional unit. Technology scaling leads to increased wire resistivity, increasing the cost of moving operands greater distances. In the general case, forwarding requires the output of any instruction executing on any functional unit to be sent to the input of any other instruction. To ensure that the dependent instruction can be begin executing on the next cycle, a fully connected network among functional units is required. The hardware complexity of these networks typically increases quadratically with the number of functional units. The average length of any link also increases linearly with the functional units.

Collectively, these overheads limit the scalability of the Harmony execution model. They are partially mitigated by increasing the granularity of kernels as the next section will show. However, the problems of kernel scheduling and dependency tracking and data-flow through variables remain important even as the model is scaled up to larger systems.

5.2 Many Core System on Chips

Many-core system on chips are composed of a network of heterogeneous processing elements. When the Harmony execution model is applied to this class of system there are several fundamental changes. The complexity of compute kernels increases, various aspects of the system such as the cache hierarchy, memory controllers, and on-chip network are shared among cores, and the runtime functionality of scheduling kernels and managing their execution is delegated to a software component running on one of the processing elements, rather than being implemented as a state machine in hardware.

5.2.1 System Overview

There are several major differences between system on chips and processor functional units. Limited capacity scratch-pad memory may be available as a shared resource among several cores, the processing elements may share an address space under varying coherence and consistency models, processing elements may have different micro-architecture or instruction sets, and kernel scheduling may be delegated to a processing element or a specially designed ASIC.

Scratch-pad Memory and Caches The purpose of on-chip memory is to provide low latency and efficient access to storage with a limited capacity. For caches, this access is managed automatically and for scratch-pad memory it is managed explicitly by the application.

A Shared Address Space One of the major advantages of a many-core system is that the individual cores are allowed to share a single address space. This has several consequences. First, explicit Direct Memory Access (DMA) copies between memory regions are not necessary; the single shared address space allows dependent kernels running on different processing elements to read the results from previous kernels

directly. Second, the problem of packing variables into memory regions is simplified because the complete memory space has not been hard partitioned into smaller sub-regions. Finally, the consistency and coherence mechanisms require attention from the runtime. Memory spaces that provide sequential consistency and coherence incur a hardware and performance overhead compared to more minimalistic schemes, but they also preclude the need for any runtime involvement. Weaker consistency models and non-coherent memory may require the runtime to explicitly flush cache lines and issue memory fence operations on kernel launch boundaries.

Heterogeneous Cores Cores are typically capable of executing arbitrary kernels through the use of dynamic binary translation or emulation techniques, although there are some fixed function ASIC cores such video decoders that may be practically incapable of executing all but a small set of specialized kernels. However, for the most part cores are capable of executing many or all kernels, and heterogeneity tends to come in the form of different micro-architectures, different ISA extensions, or completely different ISAs. Core with ISA extensions may also include complex functional units that are capable of executing well-defined instruction subgraphs. Multiply-accumulate functions are common examples of these complex functional units. Collectively, a system will be composed of heterogeneous clusters of homogeneous cores, for example a quad-core CPU coupled to a 48-core GPU. In these case, it is possible to map kernels onto clusters of cores or individual cores as long as the kernels themselves are sufficiently parallel.

Dedicated Control Is Optional Although runtime functionality of tracking kernel dependences, speculatively fetching and issuing kernels, and performing kernel scheduling is typically performed by dedicated hardware in OOO processors, for system on chips, it is possible to perform this functionality with hardware support or completely in software. The shared address space and tightly integrated nature of

these systems potentially allows them to benefit from the ability to schedule and manage kernels with lower latency than a pure software implementation. This is the most beneficial in situations where kernels are relatively simple and complete quickly. An alternative approach to adding dedicated scheduling hardware is to aggregate several kernels together such that the scheduling, launch, and synchronization overheads can be amortized over a group of kernels. This optimization is discussed in detail in Chapter 6.2.5.

5.2.2 Resource Sharing

Many-core system on chips allow close interactions between cores, which provide kernels with the opportunity to execute cooperatively or competitively. Resources are shared to a greater degree than with other classes of systems, and interactions between concurrently executing kernels become important performance considerations.

Memory Hierarchy In a many-core system, this memory may be shared among multiple cores: caches will accept memory requests from multiple cores and scratch-pad memories need to be partitioned by the application or runtime into distinct sections that can be accessed independently. Both of these cases create contention for bandwidth and capacity, which may degrade the performance compared to the case where a single core has exclusive access to the memory. These effects complicate the problem of scheduling kernels because performance models that only consider single kernels and single cores do not model the interactions between multiple kernels executing on cores that share resources.

Power and Thermal Budget Cores that share the same package also share the same power supply and heat-sink. Modern designs use dynamic voltage and frequency scaling to control the power consumption and indirectly the heat production of individual cores. It is possible for performance intensive kernels to consume an excessive

amount of the chip's power or thermal budget and cause other cores to be throttled. In these cases, system throughput may be improved by steering kernels to slower, higher efficiency cores.

5.3 Directly Connected Accelerator Boards

The next class of systems moves beyond the scope of a single package to a complete node composed of multiple heterogeneous processors that are referred to as accelerator boards.

5.3.1 Compositional Systems

These systems are constructed by piecing together a collection of modular accelerator boards, each optimized for a specific class of application. The complete system is composed of a central processing element and memory space that distributes tasks to a series of accelerator boards that are connected via a system level interconnect. Over the lifetime of the system, the modular nature of accelerator boards allows them to be upgraded with newer models as budgets permit or as new boards are developed; it also allows failing boards to be swapped out and replaced. The dynamic composition and evolution of these systems present significant challenges to many programming models and existing applications: adding a new board may change the optimal partitioning and scheduling of tasks. Similarly migrating code from one accelerator to another is difficult in the case that a new revision of an accelerator is used, or if the task is migrated to a different type of accelerator. In the Harmony execution model accelerator boards are abstracted as processing elements and the runtime can adapt the kernel schedule as the system configuration changes.

The Master CPU These systems usually are configured with a single CPU that runs an operating system and directs the execution of tasks on accelerator boards, forming a master-slave configuration. In Harmony, the CPU typically contains the

program binary and executes the runtime component that controls the attached accelerators. The relative performance of the CPU limits the complexity of the algorithms that can be used to perform kernel scheduling, although they can typically be much more heavy-weight than in a system on chip or within a single processor. There is also an issue of whether or not to allow kernels to execute on the master CPU and compete for resources with the runtime. This is related to the decision of how many hardware resources to devote to the control and scheduling logic in processors as opposed to the processing elements or functional units. However, in this situation, the ability to time-multiplex kernels and the runtime on the same CPU allows a more dynamic allocation of resources. If the CPU cores are given low priority during scheduling, the runtime will only compete with kernels for execution time when the rest of the system is saturated, and the runtime is not required to issue kernels quickly.

Accelerator Boards Accelerator boards are typically complete self contained systems including at least one but possibly many processing elements, attached memories, and network interfaces. Common accelerator boards include data parallel graphics processors, video decoders, network packet processors, FPGAs, or even more generic processors like Cell [28] or Clearspeed SIMD arrays [109].

System Interconnect The role of the system interconnect is to allow the master CPU to communicate with the accelerator boards. Historically, PCI [131] and PCIe [132] have evolved from shared buses to point-to-point links with latency and bandwidth characteristics in between on-chip and node-to-node interconnects. This makes DMA operations relatively expensive compared to memory operations within an accelerator board.

Distributed Memory A major difference between this class of system and many-core system on chips is the lack of a shared global memory space. Many accelerator

boards use incompatible memory technologies and interfaces and only support communication over system interconnect. This typically allows for explicit DMA transfers from one accelerator's memory region into another's. However, it also imposes a performance overhead in migrating a data-dependent chain of kernels from one accelerator board to another. The kernel scheduling problem is more difficult in this case because it is necessary to model the cost of DMA operations as well as kernel execution.

5.3.2 Accelerator Classes

Common sets of accelerators exist for graphics, video and signal processing, as well as programmable hardware like FPGAs.

GPUs GPUs have evolved into general purpose parallel and vector processors with specialized instruction sets, custom functional units for common complex operations, hardware management of thread creation and work distribution, and memory hierarchies optimized for long latency high bandwidth bulk transfers. GPU accelerator boards are paired to on-board GDDR memory that stresses high pin bandwidth at the cost of low latency access. GPUs rely on applications with a large number of data-parallel threads with regular control flow and data accesses to take advantage of multiple cores with wide Single Instruction Multiple Data (SIMD) units. As long as an application has these characteristics, the theoretical performance of these processors can be greater than that of OOO CPUs by a factor of 25 or more.

FPGAs FPGAs are used as accelerators because they can be quickly programmed to emulate arbitrary logical operations that would normally be implemented using gates connected via a network of metal layers. FPGAs implement the same operations using RAM-based Look-Up Table (LUT)s connected via a programmable network of muxes. Although LUTs are significantly less efficient than custom designed gates and

even standard cell libraries, the ability to quickly change to hardware that is heavily optimized for a specific operation can allow FPGAs to outperform general purpose processors when executing some classes of application. The major limitations of FPGAs stem from the fact that programming them requires as much effort as building a processor using a standard cell library, making it only feasible to run a small set of highly important kernels on them.

Video Encoder/Decoders The demand for real-time high definition video coupled to advances in video Compressor-Decompressor (CODEC) standards have resulted in the development of ASICs that are over 500x more power efficient than general purpose CPUs. These ASICs are typically implemented as state machines for each phase of the compression algorithm being used. The phases typically involve time domain to frequency domain conversion, filtering and scaling, entropy decoding, and motion vector transformations. The functional units and memory structures are typically tuned for the phase being performed and may involve complex functional units with hundreds of inputs and outputs, operand queues rather than register files that feed operands directly into functional unit inputs, staging areas between phases that do not require storing intermediate results into memory, a pipeline structure that can operate each phase concurrently, and carefully tuned circuitry designed only to hit a specific performance target.

General Purpose Accelerators Beyond GPUs, FPGAs, and video CODECs, there are several other accelerator boards that are designed with more general purpose applications in mind. Examples of these include Intel Larrabee [130], which includes a large number of lightweight x86 cores running heavy-weight pthreads, Intel IXP [2], which combines lightweight RISC cores to dedicated network interfaces and cryptography engines, and IBM Cell [28], which includes an OOO core controlling an array of SIMD cores and local store memories.

Upgrading Accelerators The modular nature of the system level interconnect allows a single system to be reconfigured with a different set of accelerators throughout its lifetime. These upgrades may occur as new hardware is acquired, new revisions of accelerator boards are developed, or as individual boards fail. For example, in the time period between 2007 and 2010, five different revisions of GPU accelerators were released by NVIDIA, each supporting a new feature set with ISA extensions for atomic operations, more registers, bit manipulation instructions, explicit cache control instructions, and new core micro-architectures. A system using GPU accelerators that was upgraded during this time may end up with several GPUs with different performance characteristics.

5.4 Heterogeneous Clusters

Moving from nodes with attached accelerator boards to clusters of individual nodes does not change the form of problem presented to the Harmony execution model. Nodes or groups of nodes can be classified as processing elements, attached per-node memory can be aggregated together using PGAS models into memory regions, and kernels can be implemented using data-parallel or message passing programming models such as UPC [23] or MPI [117].

5.4.1 Node Organization

From the perspective of the Harmony execution model, a cluster can be partitioned to groups representing processing elements with attached memory regions. The central control can be a single node control node or a distributed runtime that collectively performs kernel scheduling. These systems typically communicate through a regular interconnection network that enables data exchange and global synchronization. Memory is typically distributed in a per-node fashion. There are several unique problems associated with determining appropriate boundaries for processing elements, addressing the disconnect between interconnect and node performance, and handling

distributed memory spaces.

Heterogeneous Groups Homogeneous groups of machines from the same generation of hardware may be considered to be a single processing element, and entire clusters may be formed from a number of such groups. Groups that lose nodes or network links due to failures may also exhibit heterogeneous performance characteristics that benefit from a dynamic mapping of kernels to groups of nodes. It is also worthwhile to note that complete systems may be composed by first forming small groups of nodes into processing elements, and then recursively applying this process to build up larger groups of processing elements. To accomplish this, it is also necessary to recursively partition the kernel CFG into fused kernels that can be subdivided during execution and distributed to the processing elements within a group. This partitioning step can be done automatically by applying a graph partitioning algorithm to the kernel CFG in the Harmony compiler; it can also be assisted by the programmer if he uses a language that recursively defines compute kernels such as Sequoia [50].

Regular Interconnects Data sharing and distribution becomes more important at this level of granularity because the relative performance of individual nodes tends to scale faster than the performance of the node-to-node interconnection network. At the lowest level, data can be exchanged between function units at the same rate that it can be consumed by functional units. At the cluster level, the highest performance infiniband links implementing the EDR [62] standard can transmit data at 37.5GigaByte (GB)/s, while the fastest performing nodes with GPU accelerator boards may provide up to 10 TFLOPs; this is a factor of 266 difference. The Harmony runtime scheduler must deal with this constraint by accurately modeling the cost of relative cost data transfers and kernel execution. Furthermore, this disparity reduces the effectiveness of greedy schedules that only consider the expected benefit of moving single kernels to remote nodes. These systems may benefit from static optimizations

that vertically slice the kernel control flow graph into independent chains of kernels that do not share data between other chains, and collapsing these chains into fused kernels with high compute density.

5.4.2 Distributed Memory Spaces

The extreme scale of these systems makes it unlikely that a single runtime will be able to schedule kernels on the entire set of processors by treating them all as processing elements that are capable of executing individual kernels. Furthermore, most applications do not have enough KLP to scale beyond a modest number of processing elements. This problem is sidestepped by treating distributed systems as a collection of groups of nodes, where each group represents a Harmony processing element. A side effect of this decision is that groups of nodes will typically not have access to a single shared memory space. In these cases, the use of a PGAS model that treats a group of memory modules connected to several nodes as a single memory region or kernels that explicitly deal with distributed data are necessary.

5.5 Summary

The previous discussions demonstrate that the components in systems spanning several levels of granularity may be represented using abstractions in the Harmony execution model. Processing elements may be functional units in OOO processors, accelerator boards in single nodes, or groups of nodes in a distributed system. This section merely intends to identify several examples that may benefit from the application of the Harmony execution model. Future systems that have components that can be represented using the abstractions of processing elements, memory regions, and central control logic may also be good targets for this model.

CHAPTER VI

MODEL OPTIMIZATIONS

The previous chapters have built up to a complete description of the Harmony execution model and covered examples of systems that it can be applied to. This section introduces and describes optimizations that can be performed on the model. All of these optimizations use knowledge about the structure of the program or the capabilities of the system executing it to extract better performance or efficiency from the system; they all require a process for obtaining this information. The first section identifies relevant system and application metrics and describes techniques for measuring them. The following sections deal with optimizations that can be performed statically on the Harmony program representation as well as those that require program or system information that is only available dynamically at runtime. The final section describes optimizations that make the system tolerant to faults and component failures. For the most part, these optimizations are described in terms of the abstractions in the Harmony execution model only so that they may be applied to any of the systems described in the previous sections.

6.1 Performance Models

Performance models are concerned with identifying and measuring a set of metrics that should be maximized or minimized during the execution of a program. These metrics can be used by static or dynamic optimizations that transform the program or configure the hardware resources to, for example, reduce program execution time or minimize total energy consumption. Beyond simply measuring execution time or energy consumption, it is also necessary to derive analytical or statistical models that can be used to predict application or system behavior. Predictions are necessary for

scheduling and mapping kernels to efficient processing elements.

6.1.1 Performance Metrics

Performance metrics are the objective function being optimized by the Harmony runtime system. In many cases, the function being optimized is execution time although more recent power constrained systems are just as likely to try to minimize the energy required to execute an application. From the perspective of the Harmony runtime, the objective function does not influence the design of the system. The system merely requires it to determine the effectiveness of various decisions that are made throughout the optimization process. Most of the examples in this dissertation choose execution time as the metric being optimized, but it should be understood that this is merely a common choice and not the only choice.

Execution Time Execution time is probably the most common metric used to measure system performance. Although the primary design criteria of many systems is to return a result as quickly as possible, it tends to bias work distribution towards the fastest processors in a system and run them at full capacity, without any regards for how efficient these processors are. On the other hand, execution time is very easy to measure, as performance counters and hardware clocks are readily available on most processing elements and directly exposed to software. In the case of functional units on OOO processors, the latencies of most operations are typically deterministic or easily predictable, and even those are not such as memory accesses can be easily measured using hardware counters and timestamps.

Energy Energy expended during the execution of a kernel is significantly more difficult to measure than its execution time. Although some systems include hardware sensors that measure power consumption, these are generally not standardized

and difficult to measure in software. Analytical models that relate low level micro-architecture events to energy consumption have become increasing accurate tools for measuring energy [68].

6.1.2 The Kernel Mapping Problem

The kernel mapping problem involves determining the optimal mapping from a set of kernels to processing elements. Given a set of kernel, variables, and their dependences, kernels should be mapped to processing elements that will execute them as efficiently as possible. To perform this mapping effectively, a predictive model that determines the efficiency of executing a given kernel on a given processing element is necessary. Depending on the definition of efficiency, the model may predict the kernel execution time, the amount of energy consumed, or another representative metric. These may be specified explicitly by the application or system; they may also be observed through indirectly through measurements. In both cases, metrics are classified into static and dynamic categories indicating that they can be measured by observing the kernel or system structure off-line, or whether they change over time as the program executes.

6.1.3 System Characteristics

System characteristics are observable attributes of a processing element in a system. The combination of kernel and system characteristics are used as inputs to models that predict the behavior of a specific kernel when it is executed on a specific processing element. One difficulty in selecting system characteristics is that they are specific to the type of the processing element being modeled. The number of cores, clock frequency, average memory latency, and peak memory bandwidth may be sufficient for many coarse-grained models of execution time on a given processor. However, the core count may not make sense in the context of an FPGA, where the number of LUTs may be more appropriate.

6.1.4 Static Kernel Characteristics

Static kernel characteristics are quantifiable aspects of kernels that can be measured before it is executed. The number of loops inside the kernel or the number of input kernel variables are examples of simple static characteristics. The number of divergent branches, the ratio of uniform and affine operations to generic operations [34], the average basic block size, and the number of registers spilled are examples that require deeper analysis to obtain.

A fundamental observation is that many static kernel characteristics are strongly correlated with dynamic characteristics, enabling useful predictions about the behavior of a kernel to be made before it is executed. Figure 47 shows the principle component analysis of a collection of static and dynamic kernel metrics. The grouping of static and

dynamic metrics into the same principle component shows a high correlation. On the other hand, some metrics may seem like they could be measured statically, such as the number of static instructions, or the size of a loop body. However, dynamic translation and optimization systems may eliminate dead code or unroll loops, introducing effects that are independent from static metrics and difficult to predict. Additionally, some aspects of a kernels execution are not captured in the structure of its computation. This occurs most commonly for algorithms that are highly data-dependent. For example, the complexity of matrix multiplication generally increases with the matrix dimensions (the input dataset size). Other algorithms such as graph traversal or path-finding may have complexities that vary with the relative density or sparsity of the graph. Kernels implementing these algorithms typically require dynamic measurements of data input sizes or distributions to make accurate predictions.

6.1.5 Dynamic Kernel Characteristics

Dynamic kernel characteristics are measurable quanta that may only be obtained while the kernel is executing. They could be the number and mix of dynamic instructions, the number of memory pages accessed, the amount of data-flow through registers, or through an on-chip local store. Usually dynamic characteristics are less useful than static characteristics because i) they must be measured while the kernel is executing, and ii) they are often collected too late to influence kernel scheduling and optimization.

In many cases, dynamic measurements are more useful for determining the accuracy of outstanding predictions and refining models that are learned as the program executes than as inputs to predictive models. This is because they are usually available too late to make effective predictions. However, there are a few notable exceptions. Jiang et al. [76] show that dynamic metrics that are sampled during kernel startup are highly correlated with the same metrics during the remainder of the kernel execution. This potentially enables kernels to be run in a high overhead instrumentation mode for a fraction of their total execution time before they are re-scheduled or re-optimized using this dynamic information.

Similarly, systems that support migrating kernels between processing elements or dynamically generating kernel binaries may continuously tune the performance of the system by refining kernel schedule or slowly applying higher levels of optimization. Processing elements that support preemption may periodically recompute the schedule for currently executing kernels and shuffle mis-scheduled kernels among processing elements. In a similar manner, many dynamic compilers support regenerating the code for a given function or region with a different set of low level code optimizations. If a predictive model determines that a kernel my benefit from a new optimization pass, the kernel could be suspended, the code could be regenerated with the new pass, and the kernel could be made to resume its execution using the newly

optimized binary.

In these systems, the accuracy of a model's predictions increase over time as more dynamic information is collected. However, this information also becomes less useful over time because optimization decisions must be made at specific points during the execution of a program. For example, predicting that it will be advantageous to apply loop unrolling after the last loop is executed is not useful; it may even be harmful if it is not known that no more loops will be executed. For this reason, dynamic information that is gathered after a heuristically determined 'startup' period should not be used to influence a optimizations that are applied to that kernel. Instead, it should be recorded and used to refine predictive models for future kernels in the same application or as profiling information that is propagated between application runs.

6.1.6 Predictive Models

Once dynamic or static information is gathered from a kernel, a model is required to predict its behavior when executed on a specific processing element. Although analytical models are the most prevalent in existing systems such as OOO instruction schedulers and runtime optimizers, the increasing complexity and irregularity of large scale systems is likely to stimulate the widespread adoption of statistical models that can be derived systematically.

Analytical Models Analytical models relate static and dynamic kernel characteristics to performance metrics through a set of manually derived equations based on knowledge of a processing element's design. A simple model might combine ILP and Memory Level Parallelism (MLP) kernel characteristics with a processing element's super-scalar width, clock frequency, and memory bus frequency to predict the total amount of time required to execute a kernel. More complicated models might employ information about functional unit latencies and instruction dependences to statically schedule blocks of instructions combined with predicted loop counts to determine the

expected number of cycles required to execute a kernel. A processor timing simulator such as SESC [116], PTL-Sim [157], GPGPU-Sim [10] could also be considered to be extremely detailed analytical models, although they would not be useful for kernel scheduling or optimization due to their excessive compute complexity. A common characteristic of these models is that they make use of knowledge of the physical design of a processing element and how the abstractions used to specify kernels are mapped onto physical resources.

Statistical Models Whereas analytic models attempt to capture and faithfully model the essential processes that are used by processing elements to execute compute kernels, statistical models assume that there is an unknown relationship between kernel characteristics and a metric being predicted; they attempt to discover this relationship automatically using empirical methods. Techniques like principle component analysis can be used to identify a representative set of uncorrelated kernel characteristics. These can be coupled to clustering algorithms such as regression trees or adaptive regression splines to create a model that classifies programs into related categories and includes constructing regression models separately for each category. Machine learning techniques such as neural networks or support vector machines may also be employed to automatically construct and train models.

The main advantage of statistical models is that they can be constructed automatically without human interaction. Systems that routinely add new classes of processing elements as new hardware is released or upgrades are purchased do not require the models to be updated as well as in an analytical model. The same process can be reused to automatically build a new model for the updated system.

6.2 Static Optimizations

Static optimizations may be performed on Harmony programs stored in a binary format before they are executed. Some of these optimizations reduce the complexity

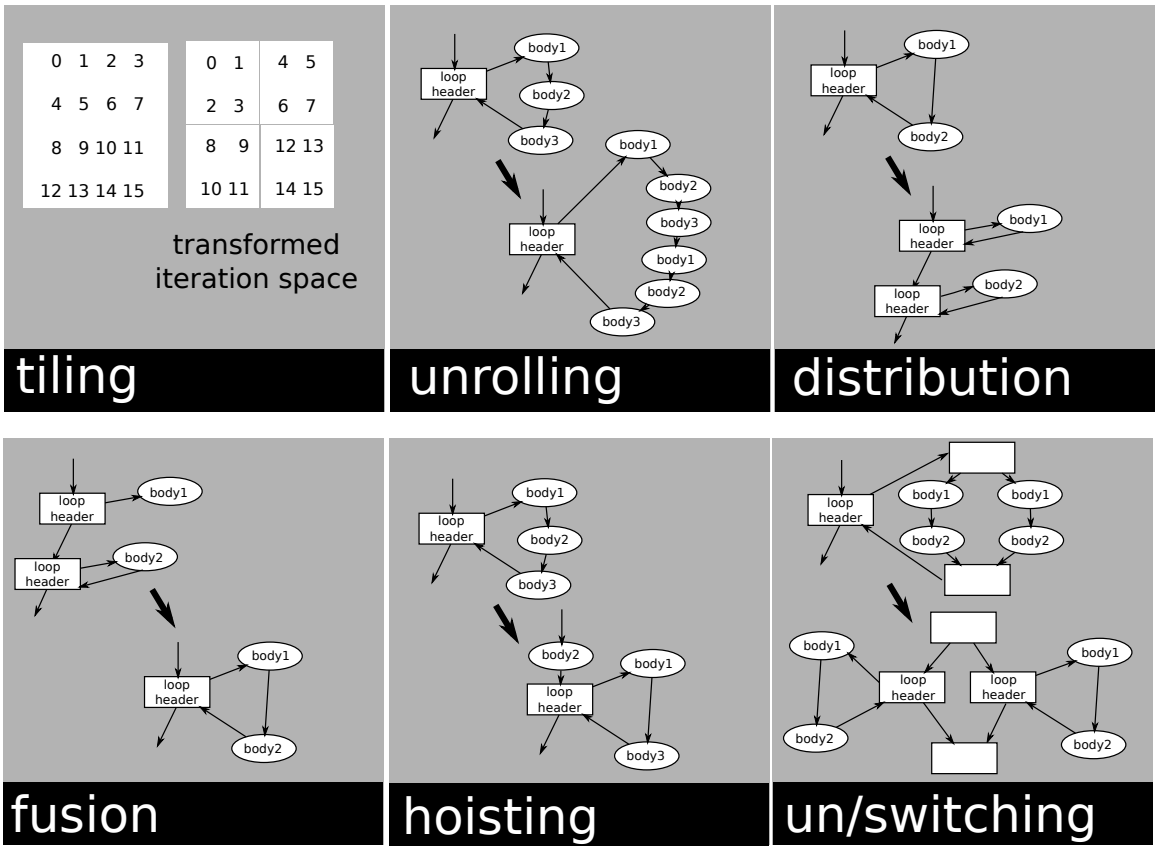

Figure 15: Common transformations on loop structures in the Harmony Kernel IR.

of Harmony programs by eliminating redundancy. Others attempt to refine or coarsen the granularity of kernels to allow for more fine-grained runtime optimizations or to reduce runtime overheads. Finally, some attempt to provide hints at the dynamic behavior of the program based on its structure. Many of these optimizations have equivalents in mid-level optimizing compilers. They are listed here to show that they can be applied directly to the Harmony IR form, and to note special considerations when dealing with Harmony programs.

6.2.1 Loop Transformations

Looping structures in the Harmony IR correspond to cycles in the kernel CFG. Loop transformations on imperative programs are well-studied, and this section shos that most common optimization can be applied to Harmony applications as well. In this

case, loops consist of chains of compute kernels terminated by control decisions.

Even though the CFG does not represent them explicitly, natural loops, or reducible loops, may be discovered from the dominator tree of the program by applying the definition of a natural loop directly, i.e. that the header of the loop dominates the exit blocks in the loop, and that all exit blocks have an edge that connects them to the header. Irreducible loops may be converted into reducible loops by various transformations such as that proposed by Zhang et al. [158]. Once loops in the program have been identified, it is possible to apply various transformations to them that trade off various parameters such as the loop overhead, the amount of work performed in the loop, the code size, the number of variables required, or the access pattern of iterations in the loop and the access patterns of the kernels contained within it. Several common optimizations are listed below:

- **Loop Tiling**: Loop tiling assumes that an analysis routine is able to statically discover the induction variable used in the loop or a series of loops that form a nest and determine the iteration space. Once this is done, the body of the loop can be examined to determine dependences that span loop iterations, for example, if a variable is produced in one iteration and used in another. At this point, the iteration space of the loop may be transformed such that all data dependences are satisfied to reduce the working set size, or block data accesses into sections that will easily fit into a cache or runtime controlled memory region. This of course assumes some knowledge of the memory regions in the machine being targeted, and may actually degrade performance in cases where the assumed machine model ends up being incorrect. It is also worth noting that tiling may be more difficult in the general case on the Harmony IR because it may be difficult to determine the behavior of kernels that modify the loop induction variables, whereas traditional compilers where kernels are instructions are typically aware of the behavior of instructions. This optimization assumes

that the compiler is able to examine the kernels updating the loop induction variables and determine their behavior to discover the iteration space. The top-left subfigure in Figure 15 shows an example of loop tiling where a row-major iteration space is transformed into a 2x2 tiled space.

- **Loop Unrolling**: Loops typically consist of a fixed amount of work that is performed each iteration, incurring a fixed cost of a kernel to update the branch condition and branch back to the beginning of the loop body. Loops also typically make subsequent iterations control dependent on some kernels in the current iteration, potentially limiting the amount of KLP that can be discovered at runtime or compile time. Unrolling replicates the body of the loop multiple times in an attempt to mitigate this overhead. In cases where the induction variable can be discovered by the compiler, it may be possible to prove that subsequent iterations will be performed and omit checks, removing control dependences. Even when this is not the case, it is typically possible to schedule kernels across multiple iterations after unrolling has been performed using modulo-scheduling [91]. The top-middle subfigure in Figure 15 depicts a loop whose body is unrolled twice and the control dependent back-edge is removed.

- **Loop Distribution**: Loop distribution is used to split the bodies of a single loop into several sections to improve the locality of the variables being accessed in each section. This occurs because the total set of variables being accessed during each iteration is reduced, and the same variables are typically reused across iterations easing the problem of packing variables into a memory region by reducing the working set size of the loop. The top-right subfigure of Figure 15 shows a loop with two basic blocks in the body. When the loop is distributed, these basic blocks are looped over one at a time.

- **Loop Fusion**: Loop fusion discovers a series of related loops with identical

or similar iteration counts and merges their bodies together to create a single large loop. This is effectively the dual of loop distribution; it reduces the loop overhead at the cost of potentially increasing the number of live variables (and the working set size) in the loop body. It is especially beneficial when a series of values are passed directly between kernels in the loop body. In this case, the values can be consumed immediately by the subsequent kernels and may not be required to be saved across iterations of the loop, whereas if the kernels were executed by a series of loops, they would be alive until the next loop was executed. The bottom-left subfigure of Figure 15 shows a series of two loops that are fused together.

- **Loop Invariant Code Motion**: In some cases, kernels in the body of a loop are neither data-dependent on other kernels in the body, nor the loop induction variable. In these cases, it is possible to remove kernels from the body and execute them before the loop is entered via a process referred to as loop invariant code motion. If the loop is guaranteed to execute at least once, then the kernel is not control dependent on the loop and can be simply removed. If the loop may execute zero iterations, then a guarding control decision is needed to represent the control dependency. If the kernel was a heavy-weight operation, hoisting it outside of the loop may significantly reduce the amount of work required to compute the same result. The bottom-middle subfigure of Figure 15 shows loop with three kernels in the body. The second kernel is determined to be independent from the iteration count and the other kernels in the loop and hoisted out of the body and placed immediately before the loop. Subsequent scheduling passes may further optimize its placement.

- **Loop Switching/Unswitching**: A special case of loop invariant code motion occurs when a control decision contained inside the body of the loop is eligible

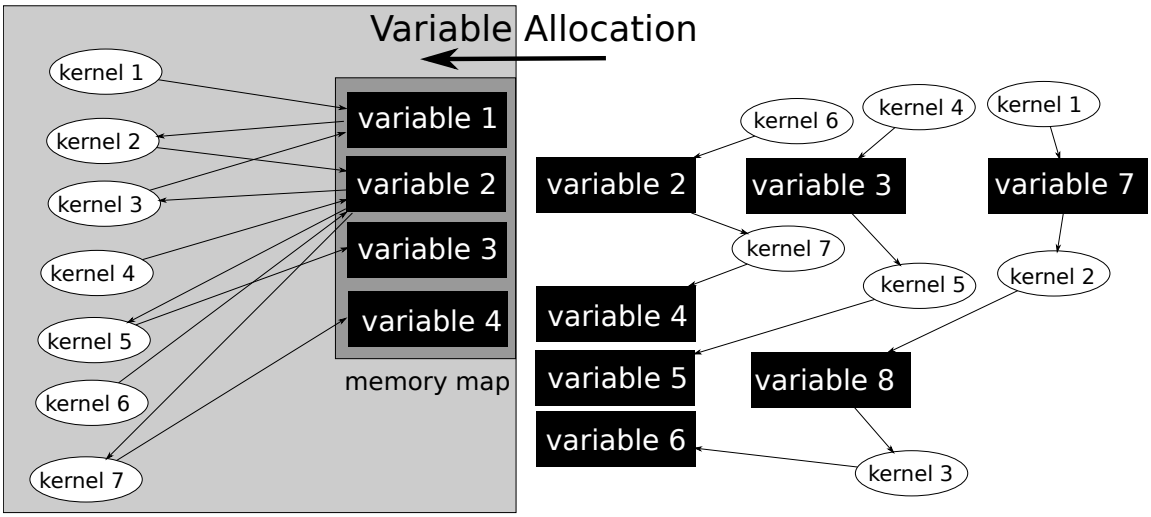

Figure 16: Variable assignment allows the compiler to trade-off space requirements and the amount of kernel parallelism in a Harmony program.

for hoisting. In this case it is possible to hoist only the control decision out of the loop. When this corresponds to a simple if-then-endif style conditional statement, the loop and its body can simply be bypassed by the control decision in the case that the condition evaluates to false. In the case when it corresponds to a if-then-else-endif statement, the loop should be distributed over the blocks corresponding to both the 'then' and the 'else' clauses, as shown in the bottom-left corner of Figure 15. In both of these cases, the overhead of the independent control decision is incurred only once, rather than on every iteration.

6.2.2 Variable Allocation

By examining the data-flow graph representation of the Harmony IR, it is possible to determine when each variable is first produced, and when it is last used. This information can be used to determine the regions over which variables need to be allocated storage in a memory region. In the case of OOO processors, registers can be reused after they are used for the last time. In the case of larger scale systems, the memory space allocated for variables can be reclaimed. It is even possible to time multiplex the use of the same variable between multiple kernels, if that variable's value

is dead at the time. Figure 16 shows an example of this process. The right half of the figure shows a dataflow graph containing eight kernels, where circles represent kernels and rectangles represent variables. The original representation of the program uses eight total variables, which may need to be kept in memory regions at the same time by the Harmony runtime. The left half of the figure shows a possible representation of the same program after variable allocation has been applied. Note that the total number of variables required drops from eight to four. As a specific example, the compiler is able to determine that variable7 is not used again after kernel2 is executed. It is able to reuse the space that was originally used for variable7 to store variable2. Note that a runtime component that merely watched the stream of kernels that were being executed would not be able to perform this optimization because it would not know that variable7 would never be used again until it was written over by another kernel, which never happens in this example.

Variable allocation can introduce overheads as well. Note that the example in Figure 16 creates false data dependences between several kernels, for example, between kernel2 and kernel5, that did not exist in the original program. An implementation of the Harmony runtime may subsequently break these dependences by performing variable renaming, but this may incur a runtime cost. Variable allocation also makes it more difficult to effectively partition and schedule kernels and variables onto different memory spaces because there is now more contention for variables. The dependency graph has more edges, making it more difficult to partition variables across memory spaces and place dependent kernels far enough apart to hide their latency.

6.2.3 Common-Subexpression Elimination/Re-materialization

Common Sub-Expression Elimination (CSE) identifies variables whose values are computed multiple times, saves them, and replaces chains of kernels that re-compute the same value with direct access to the saved copy. This optimization is useful

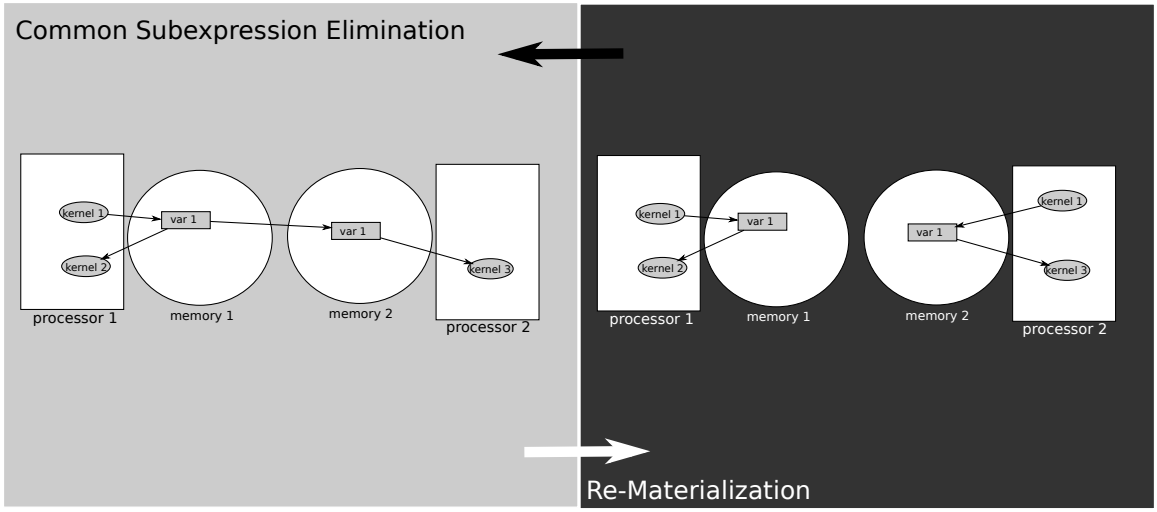

Figure 17: Navigating the space between common subexpression elimination and re-materialization allows additional storage to be allocated to hold intermediate results, or the same values to be recomputed multiple times.

because it eliminates redundant computation, and it is made possible by the deterministic and side-effect-free properties of Harmony kernels. However, it also imposes additional storage requirements by increasing the live ranges of variables, and for systems without a shared global address space, potentially requires copies between memory regions in cases where multiple kernels that access the same variable are scheduled on different processing elements. Depending on the cost of the copy operations compared to the complexity of re-executing the same kernels multiple times, CSE may be advantageous in some cases and detrimental in others. The dual of CSE is re-materialization, which identifies a series of kernels that produce the value of a variable and replicate this series in place of accesses to that variable.

6.2.4 Variable Coalescing

Variable coalescing involves merging multiple Harmony variables and replacing them with a single aggregate variable representing all of them together. The purpose of variable coalescing is to reduce the runtime overheads associated with variable renaming, dependency tracking, and potentially optimizing the movement of many small

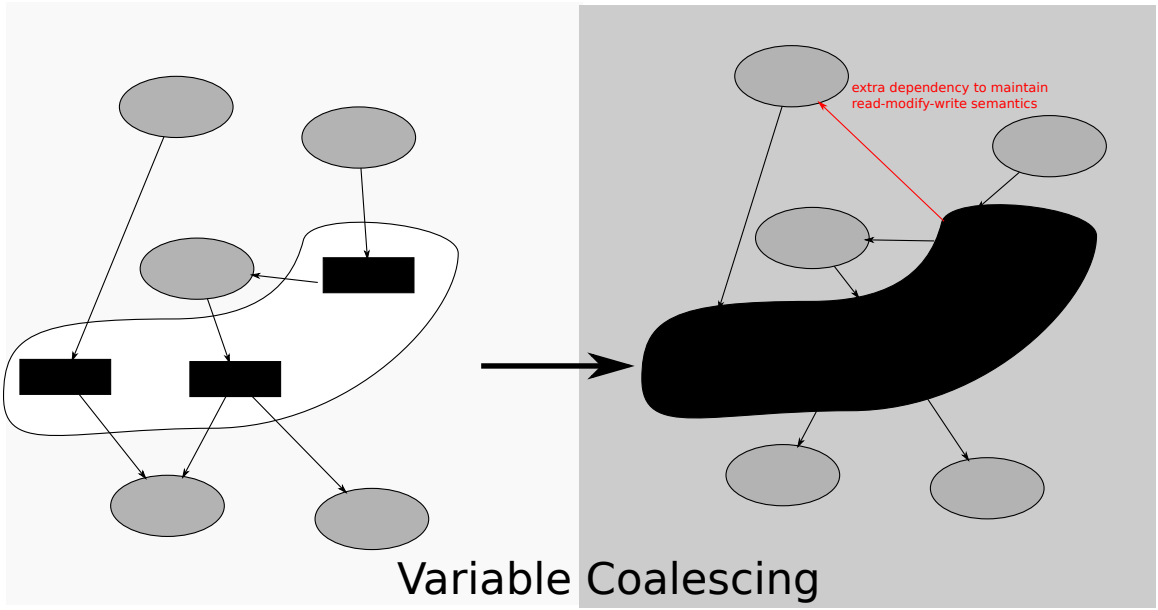

Figure 18: Variable coalescing reduces the overhead of bounds checking, and potentially improves the efficiency of a series of copy operations via batching.

variables between memory regions by combining them into a single bulk operation. Figure 18 shows an example where three intermediate variables with different access semantics are transformed into a single, coalesced, variable.

Although this example is fairly straightforward, several special considerations are necessary when performing variable coalescing:

- Output to input/output dependences: Although it would be possible for a series of kernels to produce variables using write-only semantics where the original value of the variable is discarded, coalescing changes the behavior of these kernels from updating a single variable exclusively to updating a segment of a shared variable. This requires the original values of the other segments corresponding to the other coalesced variables to remain unmodified through the execution of a kernel that writes to only one of them; it changes the access semantics of that kernel from write-only to read-modify-write, preventing aggressive optimizations that consider the old value of the variable as dead when the kernel is executed.

- Offset adjustment: It is no longer sufficient to simply pass in an opaque pointer, reference, or handle to a kernel to represent coalesced variables because they actually represent several smaller variables. In these cases it is necessary include an indexing mechanism that identifies the appropriate section of data within the coalesced variable. This effectively converts a single variable into a more complex data structure (a tuple). A runtime may transparently handle this without exposing it to the application by storing coalesced variables as a vector or list of generic variables and a list of associated offsets.

It is also possible for variable coalescing to degrade performance and the precision of other analysis. Performance may be impacted if extra data is copied during a migration between memory regions, a renaming or check-pointing operation is performed, or a variable is resized (which will trigger a resize of the collection of variables rather than a single variable). Additionally, memory alignment restrictions may increase the total size of the coalesced variable by preventing their associated data to be densely packed in cases of many small variables with strict alignment requirements. The precision of bounds checking is also affected by variable coalescing. It may be more difficult to detect a kernel that makes an out-of-bounds memory access because it may fall into the address range of another variable that was coalesced into the same data structure, potentially leading to security risks, data corruption, or incorrect execution if the application expected an error to be detected. These potential issues make variable coalescing most advantageous as a reversible transformation that does not discard the original set of variables.

6.2.5 Kernel Fusion/Fission

It can be the case that an entire Harmony program contains an uneven mixture of highly complex kernels and lightweight simple kernels. These cases may incur significant runtime overheads when executing lightweight kernels, or perform too little

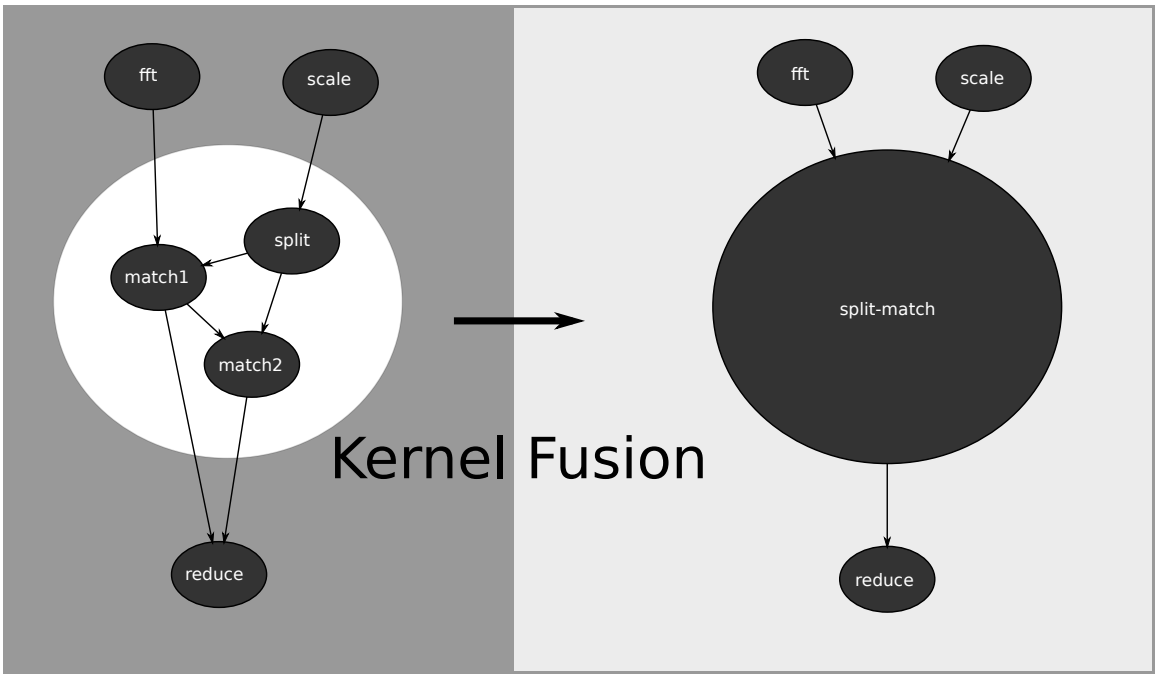

Figure 19: An example of kernel fusion, three simple kernels are merged into one larger kernel and their dependences are combined.

effort when optimizing the execution of important heavy-weight kernels. They also increase the relative importance of the performance models used to determine the cost of executing one kernel compared to another because inaccurate predictions that result in poor scheduling decisions can have more dire consequences than cases where all kernels are relatively equal. To give a specific example, consider a system with two identical processing elements, and program with one hundred small kernels that take one unit of time to execute, one large kernel that takes one hundred units of time to execute, and no dependences among any kernels. In this example, any schedule that does not place all one hundred small kernels on the same processor and the one large kernel on the other will create a load imbalance. Simple runtime schedulers that make locally optimal decisions, for example, placing small kernels on alternating processors one at a time until the large kernel is encountered, will perform poorly on such highly imbalanced problems. More sophisticated scheduling heuristics may incur significant performance overheads, and simpler heuristics such as the critical path method or list

scheduling require the window of kernels being scheduled to contain the long running kernel to be effective, implying a large fetch window for kernels. These problems can be alleviated by performing transformations statically that attempt to regularize the complexity of compute kernels.

One such transformation is kernel fusion. In the context of the previous example, kernel fusion could identify the one hundred lightweight kernel and fuse them into a single heavyweight kernel that took the same one hundred units of time to execute as the other heavyweight kernel. This significantly eases the burden placed on the runtime scheduler by reducing the number of kernels that need to be considered, as well as making the static distribution of work among kernels more uniform. In this example, even the simple locally optimal scheduler would produce an ideal schedule for the two kernels.

Kernel fusion relies on a static performance model that is able to determine the relative complexity of kernels in the Harmony IR. The development of such models is described in greater detail in Section 6.1, but the general approach is to identify strongly connected subgraphs of the program CFG with complexity that is significantly less than the average kernel complexity in the program and fusing them together into a single kernel. During the fusion process, all of the data dependences are merged. If any kernel writes to a variable then the fused kernel writes to that variable. If any kernel reads from a variable, then the fused kernel also reads from that variable. If different kernels read and write to the same variable, then the fused kernel both reads and writes that variable. Control dependences complicate the fusion process, and it is illegal to merge two sets of kernels, one of which is control dependent on a given branch and the other which is not. To fuse the program in this case, it is necessary to create multiple paths to the fused kernel, convert the control dependency into a data dependency by setting a predicate variable depending on the path taken to the fused kernel, and create a control decision inside the fused kernel

that checks the predicate and branches to the appropriate set of internal kernels.

Once a set of kernels have been identified for fusion, it is necessary to actually combine them. This can be performed via creating some aggregate representation of a kernel as a collection of other kernels, or by actually merging the low level implementations of the kernels. For example, if the compiler is aware that all kernels are actually implemented using the same virtual instruction set, Low Level Virtual Machine (LLVM) or PTX for example, then it can simply create a new function that calls each kernel in a predefined order and allow subsequent low level optimization passes such as in-lining and redundancy elimination to tune their combined implementation.

Finally, it is worth noting that although kernels that are fused together are treated as a single unit by the Harmony runtime, it is possible that during the execution of those fused kernels, they are unpacked and scheduled using a local scheduler, which may be another instance of the Harmony runtime operating at the granularity of a single processing element. The case where a set of kernels are first scheduled on a system of accelerator boards, each of which supports OOO execution is an example of such a system where high level kernels are fused sets of instructions. There is nothing preventing the recursive application of this technique to handle systems at different granularities.

6.2.6 Kernel Dependency Graph Partitioning

A related problem to kernel fusion is that of dependency graph partitioning. Rather than actually merging a subgraph of kernels into another logical kernel, graph partitioning attempts to group kernels together and provide hints to the Harmony runtime that they should be grouped together onto the same processing element and that associated variables should be placed in the same memory region during scheduling. This optimization is useful because it is easier to justify heavy-weight analysis (graph partitioning is NP hard) that has a global view of the entire program statically than

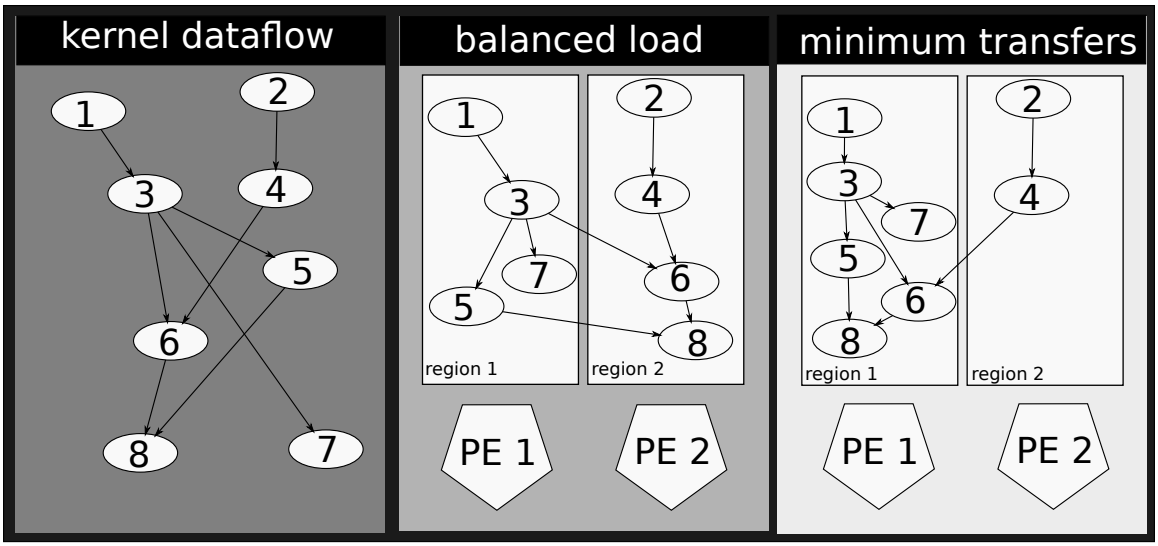

Figure 20: Statically partitioning the kernel data-flow graph allows heavy-weight partitioning techniques to optimize for load balance (middle section), memory traffic (right section), or other metrics without incurring the overheads of an NP hard problem at runtime.

runtime analysis that may be costly or difficult to perform. The problem is the traditional definition of graph partition, given a control and data flow graph of kernels, create N subgraphs such that the union of all graphs represents the original graph and some cost metric is optimized. In this case, the cost metric could be the weighted-sum of edges that span subgraphs (to reduce the number of variables transfered between memory regions), the complexity of kernels in each subgraph (for load balancing), or some other metric. Figure 20 shows an example of a kernel dependency graph that is partitioned using each of these metrics.

Implementation or Target Bias Once a program has been partitioned into subgraphs, it is also useful to influence their mapping onto specific processing elements using information from static performance models. In highly heterogeneous systems it is advantageous to form subgraphs from kernels that can be efficiently executed by the same type of processing element and passing this information on to the runtime. This information can be used to augment decision models in the runtime with more

heavy-weight models that can draw from profiling information or a database of a large number of applications statically.

6.2.7 Static Kernel Scheduling

Although Harmony applications specify their data and control dependences explicitly in the program IR, it is also possible to rearrange the sequential order of kernels to influence the scheduling order at runtime. If the static compiler or optimizer has information about the expected machine configuration, such as the expected number of processing elements, it can pre-schedule the kernels in the Harmony IR, for example, by placing kernels with back-to-back dependences as far apart as possible.

6.2.8 Static Predictions

The final set of static operations is concerned with embedding performance hints about the expected performance of kernels on different types of processing elements statically. In the general case, kernel complexity is a function of static and dynamic parameters. However, in some cases a purely static model may capture enough information about a computation to be useful. Specifically, a performance model that is parameterized on a combination of machine, static, and dynamic parameters may be aided by explicitly recording the static metrics and making them available to the runtime through meta-data; baseline predictions that are made using only static parameters can also be included and used for initial scheduling and runtime decisions before dynamic information is available.

Static predictions are most relevant for structured, deterministic kernels.

6.3 Dynamic Optimizations

Dynamic optimizations are intended to be performed on a Harmony program as it is being executed using information that is only available at runtime. Static optimizations may also be applied in a limited capacity at runtime for programs that were not

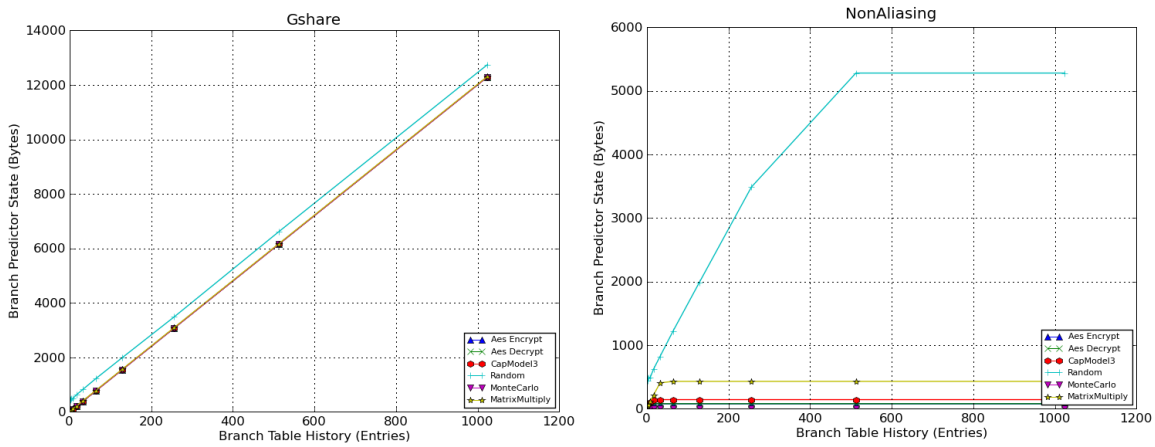

Figure 21: Hardware branch predictors typically require fixed table sizes. Software implementations may dynamically grow the size of the table to fit the demands of the application. Different applications have significantly different demands on the table.

optimized beforehand. This is useful in the case of legacy applications or lazy optimization that is only applied for programs that are actually executed. In this case, knowledge that the program is actually going to be executed is dynamic information. A slightly more detailed version of this is referred to as hot-path optimization, where static optimizations are applied to regions of a single application as they are executed.

Beyond applying static optimizations dynamically, there are several optimizations that require the use of runtime information. Examples of these optimizations include speculative execution, scheduling optimizations that rely on dynamic predictions, and variable renaming.

6.3.1 Speculation

Speculation involves performing some update to the state of the system without waiting to make sure that the update may be performed, but still making sure that it is possible to detect errors when they occur and clean up after them. This is well-defined in terms of the Harmony execution model, because kernels have deterministic effects on the system: they update only a specific set of variables. This makes it relatively easy to track the set of variables that are modified by a kernel to keep a

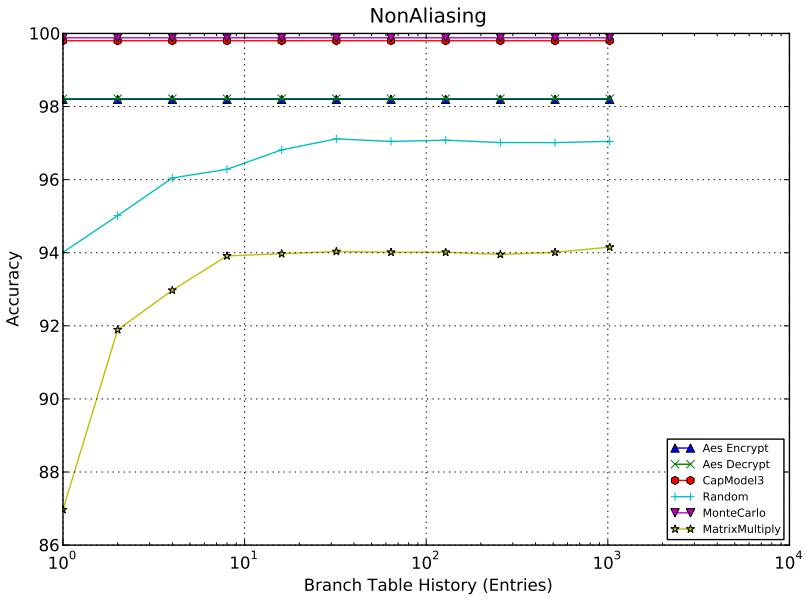

Figure 22: The difficulty of predicting branches remains relatively constant when increasing the granularity of kernels.

separation between speculative and non-speculative state. Kernels in Harmony must maintain a side-effect-free property, which significantly eases the implementation of speculation compared to other kernel-based representations.

There are several techniques that may be employed to track speculative state and roll back when a mis-speculation occurs. One possible approach is described as a case study in Chapter 8. Others include the Harp processor's approach of propagating a poison flag to all variables that are modified by speculative kernels and then tracking them down and destroying them when mis-speculation occurs [136], creating a checkpoint of the variable state before the first speculation and reverting back to it after a mis-speculation is discovered [100], or buffering modifications to memory until kernels become non-speculative [54].

In order for kernels to become speculative, it is necessary to make some prediction that affects the correctness of the results generated by the kernel. The most common case of this is used to break true data and control dependences. Branch prediction is used to determine the result of a control decision before it completes execution,

and data prediction is used to predict the final value of a variable before a kernel that is writing to it finishes execution. This can be accomplished by simply picking a random target of a control decision, or by predicting, for example, that a previous kernel will not actually modify a variable even though it lists a write operation on that variable. Of course, more sophisticated predictions are usually used in practice.

Branch Prediction Branch prediction involves determining the target of a control decision before it has finished executing. Kernels that are control-dependent on the branch may be launched speculatively, and mis-speculation are detected when the branch resolves. This is typically the easiest form of prediction due to code reuse in looping structures and correlations between nearby control decisions created by hot-paths through the program. Even simple prediction schemes using hardware history tables [101,142] can achieve over 97% accuracy for many types of applications. The most common form uses a series of saturating counters, one for each branch, that records whether the branch was taken or not taken when it was previously encountered. Simply predicting the same target as the last instance of the same branch is enough to capture the behavior of many common program structures, mainly loops. Another common form involves recording a set of common paths through the program, which can be accomplished simply by saving a list of targets of control decisions of at most N targets. Complete predictors can be formed by combining both of these individual predictors with a meta-predictor that tracks the predictor that has had the most success for each branch, and to use predictions from that predictor. These schemes are typically implemented completely as memory tables in hardware for predicting branch instructions, and they can also be implemented as data structures in a software implementation of the Harmony runtime. Figure 21 compares the memory requirements of a software table that allocates entries lazily for branches as they are encountered, and a hardware table with a fixed number of entries.

The software implementation can achieve the same accuracy with significantly fewer dynamic entries.

Value Prediction Value prediction attempts to break data dependences between kernels by predicting the result generated by a previous kernel. Predicting values is typically much harder than branches because the space of possible values is significantly larger than the set of possible targets of each branch (especially in the case that variables may be large data structures). It is also relatively difficult to verify that a mistake occurred, since checking the contents of a large variable could be significantly more complex than checking a branch target. However, there are some specialized cases where value prediction is effective. One common case includes speculating that a kernel will not modify a variable that it lists as a writable parameter. Value prediction works in this case because some kernels specify write dependences conservatively. If the variable will be conditionally written to, then value prediction is only required to predict the internal kernel condition to predict the final value of the variable and break true data dependences between kernels. Other forms of value prediction are significantly less accurate and typically rely on the data distribution of kernel results, where some values (such as zero) are significantly more common than other values. In all cases, a history based meta-predictor is typically used to determine those kernels for which value prediction is likely to be successful.

6.3.2 On-line Machine Modeling

On-line machine modeling involves pushing the creation, rather than the use, of static and dynamic kernel models into the Harmony runtime. The same techniques that can be used off-line with profiling data can be used to augment existing models with additional information that is collected at runtime. This actually requires complex machine learning or statistical regression operations to be performed by the Harmony runtime that are is only feasible at high levels of kernel granularity to avoid excessive

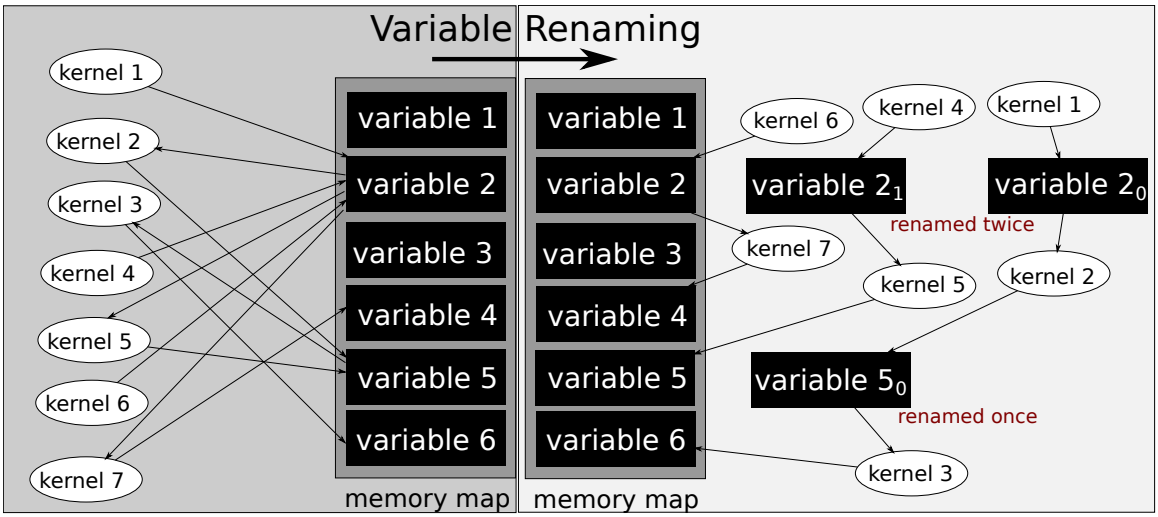

Figure 23: Variable renaming duplicates variables with a copy-on-write mechanism to eliminate false data dependences.

overheads.

6.3.3 Variable Renaming

Variable renaming is the process of creating new variables dynamically at runtime and selectively binding them to kernel input and output parameters such that the correct behavior of the program is maintained. This operation is possible on every kernel output parameter (but not input/output parameters) by allocating a new variable to hold the kernel's result and eliminating false data dependences. It is also possible for kernel input parameters by copying their value and moving them to another memory region to support executing two kernels that simultaneously read from the same variable on different processing elements. In OOO processors, this is the same as register renaming [146]. In larger systems, this is similar to applying selective copy-on-write semantics to variables before kernels that modify them are allowed to execute. Figure 23 shows an example of this process. The left half of the figure shows a series of kernels and their dependences (represented by edges). In this example, kernel1 writes to variable2, followed by a read by kernel2 from variable2, and a write by kernel4 to variable2. The value generated by kernel1 should be passed to

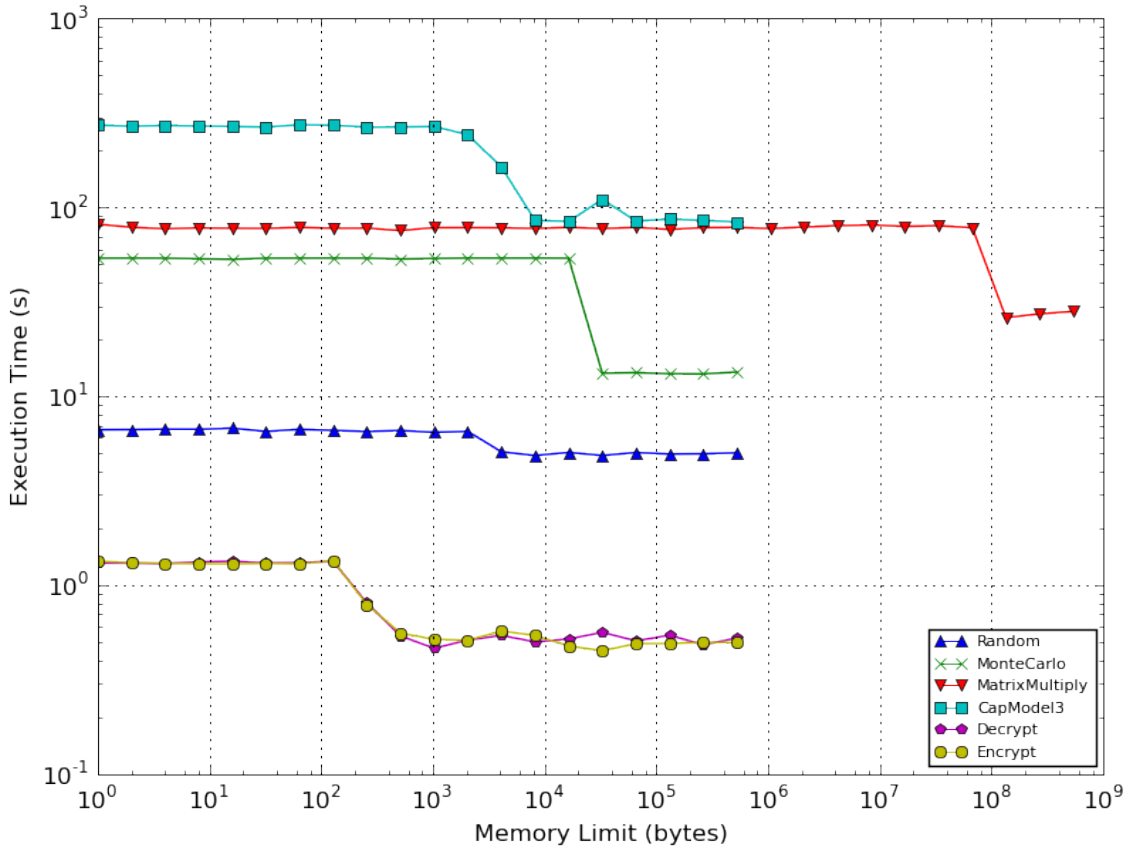

Figure 24: This figure shows examples of applications that experience a speed-up due to increased parallelism as a larger pool of memory is made available for renaming.

kernel2; kernel4 generates a new value and simply uses variable2 as storage for it. It is possible to rename variable2 immediately before kernel4 executes to allow kernel2 to execute independently from kernel4. The right side of the figure shows the same example after variable renaming. Note that variable2 has been re-allocated after the read from kernel2 into a new variable, $variable2_0$. This process is also repeated before the write to variable2 by kernel6 and the write to variable5 from kernel5 resulting in three independent chains of kernels that may be executed in parallel.

Notice that the process of variable renaming expands the total number of variables used by the program from six to nine, illustrating a basic trade-off involved in variable renaming: renaming trades off memory footprint for parallelism. Experimental results from six example Harmony applications shows this behavior in Figure 24. The

Harmony runtime was modified to create a pool of memory available for renaming. Renaming would only be performed if there was enough available memory to hold the duplicated variables. This figure shows the execution time of the kernels on a four-core CPU as the amount of memory in this pool is increased. For each of the tested Harmony applications, execution time decreases as more memory is made available, but the benefits saturate after enough parallelism is exposed to utilize all four cores. Systems with memory regions with extra space after all existing variables have been allocated may employ renaming to convert this extra space into more parallelism.

6.3.4 Kernel and DMA Scheduling

Dynamic scheduling involves mapping kernels to processing elements and variables to memory spaces as the program is being executed. The Harmony runtime performs this scheduling operation as a window of kernels and variables is fetched from the program. The scheduling component is designed under several constraints:

- it must operate on a window of kernels that is continuously updated.
- kernel model accuracy or precision may be limited.
- models may require a significant amount of time to compute a prediction.
- some information may become available late in the scheduling process. For example, the data distribution or sizes of input variables may not be known until previous kernels generate them.

The longer a scheduler waits to make a decision, the more information it has to make a better decision. At the same time, when resources become free, not making a decision immediately results in wasted processing element resources. This problem shares some attributes with other types of systems, for example, an operating system scheduling processes on multiple cores or a lightweight runtime scheduling tasks on

worker threads, and the generalized form is similar to the classical formulation of *job shop scheduling* [52] (an NP-complete problem). The major differences in this case stem from more dynamic information about kernel dependences and access patterns being available at runtime as well as the heterogeneous nature of processing elements.

Cost Models Under normal operations, it is assumed that a model of the cost of executing a kernel on a specific processing element will be available. Additionally, executing a kernel on a processing element that is not directly connected to the memory region containing the current copy of a variable will incur costs associated with explicitly allocating and copying the variable to the new space. Executing a kernel on a processing element for which code has not yet been generated may incur a dynamic compilation cost the first time it is executed. In some cases, it may also be possible to suspend kernels that have already started execution, and migrate or kill and restart them on another processing element. A cost will be incurred for a migration that may be different than for a kill and restart. Collectively modeling all of these costs can influence the decisions made by a scheduler.

Greedy Scheduling Greedy algorithms simply assign the next kernel in the dependency graph to the next available processor, choosing the most efficient processor when multiple processors are available. This scheme requires very little information and can be completed quickly. However, it also gives kernels that are encountered first significantly higher priority during scheduling than kernels that are encountered later leading to cases where a highly important processor is given to a long running kernel with a slight efficiency advantage initially and that remains unavailable for subsequent kernels that benefit from it more significantly. Several other problems occur, for example, assigning kernels to extremely slow processors when faster processors are unavailable for even a very short amount of time.

Job-Shop Scheduling Although there are some differences between the kernel scheduling problem and classical job shop scheduling, the existence of simple algorithms with strong theoretical bounds may make it desirable to cast the kernel scheduling problem as job shop scheduling and apply an existing algorithm directly. To express the kernel scheduling as job shop scheduling, it is necessary to treat all processors as equal. In this case, a predictive model should be used to determine the execution time of each kernel on each processor. Other information, such as kernel dependences and the ability to kill and restart kernels should be ignored. Kernels should be fed to the scheduler one at a time as their dependences are resolved. Simple examples of job-shop scheduling include the well-know list scheduling algorithm, where a list of jobs executing on each processing element is maintained and the next job is added to the list such that after being added, the makespan of the schedule will be minimized [56]. The current state of the art algorithm with the best lower and upper bounds involves maintaining two separate sets of processors, one highly utilized set and one lightly utilized set. Jobs are scheduled on the lightly loaded processors if the processors maintain the lightly loaded property after scheduling, otherwise they are scheduled on the heavily loaded processors [3].

Criticality Based Scheduling Another approach to scheduling involves identifying the longest or most critical paths through the kernel dependency graph, and giving them priority during scheduling. A simple algorithm involves assigning weights to kernels, then traversing the dependency graph and assigning the sum of weights of all reachable kernels to the current kernel. Once computed, the final weight ranks kernels by the amount of work that they will 'free up' after being completed. Once this is done, kernels with greater weights can be given priority during scheduling. The problem with this approach is that computing the kernel weight requires traversing many paths through the dependency graph, which can require traversing as many as

N sub-paths in highly imbalanced graphs. This may be feasible for common graphs of moderate size. However, performance constrained schedulers may require a simpler estimate of the weight of each kernel. Subramaniam et al. [140] show that the **node degree is a strong predictor of kernel criticality** based on the observation that a kernel with many dependent kernels typically creates a fan-out effect that reaches $N_{degree} * levels_{avg} * N_{avg\_degree}$ dependent kernels.

Work Stealing A more dynamic approach to scheduling involves allowing the schedule to be refined over time as more information is gathered, for example, after currently executing kernels complete and their true execution time becomes known, a load imbalance problem may occur. A simple solution to this problem is for the scheduler to attempt to remove already scheduled tasks from highly loaded processing elements and re-schedule them available processors. This is similar to existing algorithms for work-stealing [16] and typically involves picking the most highly utilized processor, removing a kernel from it, and then re-scheduling it to minimize the total execution time of the system. This should ideally take into account the overheads associated with moving associated variables to new memory spaces, recompiling kernels, or killing and restarting kernels. The general philosophy is more useful than the specific case of work-stealing: scheduling decisions should be made as quickly as possible to avoid under-utilizing resources, even if there is a lack of good information about kernel execution time or dependences. As more information is obtained, the schedule should be continuously re-evaluated and individual decisions can be changed if they improve the total system execution time given that there may be additional costs involved in reverting a previous decision.

6.4 Fault Tolerance

Although most computer systems are designed with performance or efficiency constraints, systems that are used in situations where failures have dire consequences

warrant devoting additional resources to providing a level of fault tolerance. These situations can range from downtime that incurs a financial cost, to critical failures that may potentially cause damage or injury. This section briefly discusses some possible uses of the abstractions in the Harmony execution model to continue to operate as transient errors and hard faults occur in the system. Kernels may be treated as atomic operations that modify an explicit set of variables; they create natural boundaries to perform correctness checks at runtime.

6.4.1 Kernels As Transactions

The atomic property of kernels creates explicit boundaries at which the state of the system is well defined. Specifically, executing a single kernel transforms the state of all output variables from the current state to a new state. After a kernel has been executed, it is possible to determine exactly which variables were modified. This creates a fault isolation property; all faults that result from incorrect computations by the kernel can only affect the state of variables that were modified by that kernel. It also eases the development of check point and restore mechanisms because only the previous values of kernel output variables need to be saved before the kernel is executed. Finally, it simplifies the development of modularly redundant [97] and quorum systems [126] because multiple implementations of the same kernel can be made to execute in parallel by duplicating, separating, and eventually comparing their output variables.

6.4.2 Memory Versioning

The execution of a complete application can be viewed as a series of transitions between different versions of memory, where each version corresponds to the memory state of the system before each kernel is executed. The execution of a kernel increments the version of the complete set of system memory. The deterministic property of kernels ensures that executing them same kernel on the same version of memory

will always produce the same new version of memory. This concept can be extended to individual variables by assigning a version to each one. In this case, executing a kernel will change the state and advance the version of all variables that are bound to its output parameters. Reverting the execution of any kernel simply involves restoring its output variables to their previous version. This natural bounding of the effects of executing a kernel to a contained set of variables on coarse-grained kernel boundaries creates a fault isolation property of Harmony program. A fault in a single kernel will only affect its output variables, and the previous version of those variables create a well defined point at which the application can be restarted once the fault is detected. Fault detection can be done on kernel boundaries, and if explicit memory checking is performed, it only needs to run over the set of modified variables.

6.4.3 Heterogeneity and Diversity

The heterogeneous nature of processing elements coupled with the ability to execute the same high level representation of a kernel on different types of processing elements can be used to improve the fault tolerance of the system by increasing the diversity of redundant components. Coupled with memory versioning and treating kernels as atomic transactions, it is possible to create a quorum out of many different types of processing elements. This diversity makes modular redundancy schemes more effective because it is less likely for different implementations of the same specification to fail in a way that produces identical results. When performing kernel scheduling in conjunction with a quorum system, the scheduler should be biased such that kernels are executed on a diverse set of processing elements, even if this incurs a performance penalty by gating the completion of a kernel to that of the slowest processing element. This creates an interesting trade off between performance and resiliency that can be navigated at runtime by the Harmony scheduler. This topic is not explored in depth in this dissertation; it is included to suggest a potentially fruitful area for future work.

CHAPTER VII

THE KERNEL EXECUTION MODEL

This chapter focuses on the problem of efficiently mapping a single kernel representation to heterogeneous processing elements. The solution adopted in this dissertation involves the specification of a computation in a highly structured, highly parallel, form coupled to an automated translation system that maps these abstractions onto hardware through serialization and partitioning transformations. Kernels in the Harmony execution model define atomic and opaque operations. The base model itself does not define the form of the computation contained within a kernel. Kernels can be implemented by another Harmony program at a different level of granularity, or a completely different execution model. Complete Harmony applications may in fact be composed of kernels that are implemented using a different execution model. The only requirement of the model is that there exists a mapping or translation function that allows each kernel to be executed on at least one processing element in the system.

The use of different execution models for each kernel significantly complicates the problem of executing on heterogeneous systems by requiring a separate translation function for each unique combination of processing element type and kernel execution model. Defining a single execution model that is used for all kernels in a Harmony program requires only a single translation function for each processor type. This chapter explores the development of such a kernel execution model for the specific case of heterogeneous system on chips or systems with heterogeneous accelerator boards. This choice was made out of a desire to prototype a real implementation of a structured kernel representation and a dynamic translation system, and the

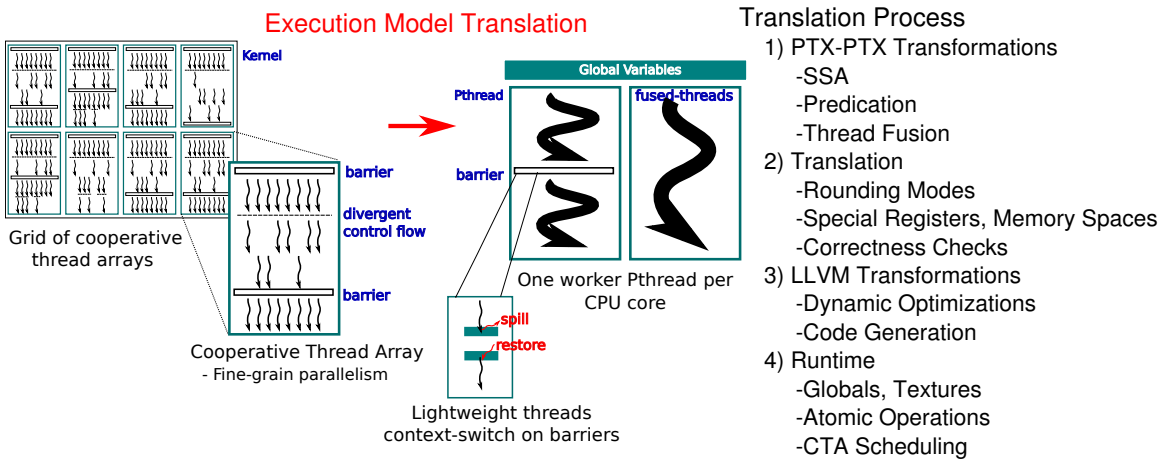

Figure 25: An overview of the translation process from PTX to multi-threaded architectures.

reader should keep in mind that this general approach is a applicable to other classes of systems as well. For example, by allowing the same representation of a multiply instruction to be mapped onto functional units with different circuit implementations.

7.1 Structured Parallelism

7.1.1 Explicitly Parallel Models

In the case of a kernel execution model, there is a question about whether or not a single representation can perform well on completely different classes of processing elements and different generations of the same processing elements. The assumption made in this dissertation is that the complexity involved in building highly optimized processing elements for different tasks will limit the types of processors in a system compared to the number of regular and modular resources that can be used to compose a processing element. Designing a few modular components and then replicating them is significantly easier than designing many different components, and that systems will be composed of a layer of highly regular, highly structured, modules that collectively perform a higher level task. Heterogeneity will also be present in these systems to optimize for specific cases, but its use will be limited compared to the modular components due to the increased complexity of its design.

Based on this assumption, the complete execution model includes two levels of hierarchy. A structured, explicitly parallel representation of a program is adopted for kernels and mapped onto regular components within a processing element while the dynamically scheduled Harmony runtime is used to manage heterogeneous processing elements. PTX is an existing explicitly parallel and highly structured execution model that is explored in the following sections as a candidate for the kernel execution model when processing elements themselves are composed of regular arrays of modules such as cores or functional units.

7.1.2 Bulk-Synchronous Parallel Kernels

This section addresses the problem of compiling explicitly parallel programs to many-core architectures. The approach advocated in this dissertation is to 1) start with a highly parallel, architecture independent, specification of an application, 2) perform architecture specific, parallel to serial, transformations that fit the amount of parallelism to the resources available in hardware, 3) generate code and execute the application utilizing all of the available hardware resources, and 4) collect performance information at runtime and possibly apply different transformations to under-performing applications. It is shown in Figure 25. The implementation described in this section, Ocelot, begins with pre-compiled PTX/CUDA applications and targets CPUs with multiple cores, a shared global memory space, coherent caches, and no on-chip scratch-pad memory.

This section covers the salient of features of NVIDIA's Parallel Thread Execution (PTX) ISA that make it a suitable intermediate representation for many-core processors as well as related work that influenced the design of Ocelot.

7.1.3 A Bulk-Synchronous Execution Model

One could speculate that PTX and CUDA grew out of the development of Bulk-Synchronous Parallel (BSP) programming models first identified by Valiant [148].

PTX defines an execution model where an entire application is composed of a series of multi-threaded *kernels*. Kernels are composed of parallel work-units called Cooperative Thread Arrays (CTAs), each of which can be executed in any order subject to an implicit barrier between kernel launches. This makes the PTX model similar to the original formulation of the BSP programming model where CTAs are analogous to BSP tasks.

The primary advantage of the BSP model is that it allows an application to be specified with an amount of parallelism that is much larger than the number of physical cores without incurring excessive synchronization overheads. The expectation is that global synchronization (barriers) will eventually become the fundamental limitation on application scalability and that their cost should be amortized over a large amount of work. In the case of PTX, a program can launch up to 2^{32} CTAs per kernel. CTAs can update a shared global memory space that is made consistent at kernel launch boundaries, but they cannot reliably communicate within a kernel. These characteristics encourage PTX and CUDA programmers to express as much work as possible between global synchronizations.

As a final point, PTX extends the BSP model to support efficient mapping onto SIMD architectures by introducing an additional level of hierarchy that partitions CTAs into threads. Threads within a CTA are grouped together into logical units known as *warps* that are mapped to SIMD units using a combination of hardware support for predication, a thread context stack, and compiler support for identifying reconverge points at control-independent code [51]. In contrast with other popular programming models for SIMD architectures that require vector widths to be specified explicitly, the aforementioned techniques allow warps to be automatically mapped onto SIMD units of different sizes. The next section briefly revisits the topic of how these abstractions, which were intended to scale across future GPU architectures, can be mapped to many-core CPU architectures as well.

7.1.4 The PTX Model

PTX is a virtual instruction set that explicitly includes semantics for management of parallelism and synchronization. Rather than requiring multiple kernels execute on different cores in the same processor, PTX allows applications to be specified in terms of a large amount of parallelism that is intended to fully utilize the resources of a single parallel processor. The basic unit of execution in PTX is a light-weight thread. All threads in a PTX program fetch instructions from the same binary image, but can take distinct control paths depending on the values of pre-set id registers with unique values for each thread. The first level of hierarchy in PTX groups threads into SIMD units (henceforth warps). The warp size is implementation dependent, and available to individual threads via a pre-set register.

In order to support arbitrary control flow as a programming abstraction while retaining the high arithmetic density of SIMD operations, NVIDIA GPUs provide hardware support for dynamically splitting warps with divergent threads and recombining them at explicit synchronization points. PTX allows all branch instructions to be specified as divergent or non-divergent, where non-divergent branches are guaranteed by the compiler to be evaluated equivalently by all threads in a warp. For divergent branches, targets are checked across all threads in the warp, and if any two threads in the warp evaluate the branch target differently, the warp is split. PTX does not support indirect branches, limiting the maximum ways a warp can be split to two in a single instruction. Fung et. al. [51] show that the PTX dynamic compiler can insert synchronization points at post-dominators of the original divergent branch where warps can be recombined. Some of my prior work [43] provides an alternate method that identifies the earliest possible point where warps can be recombined.

As shown in Figure 26, the second level of hierarchy in PTX groups warps into concurrent thread arrays (CTAs). The memory consistency model at this point changes

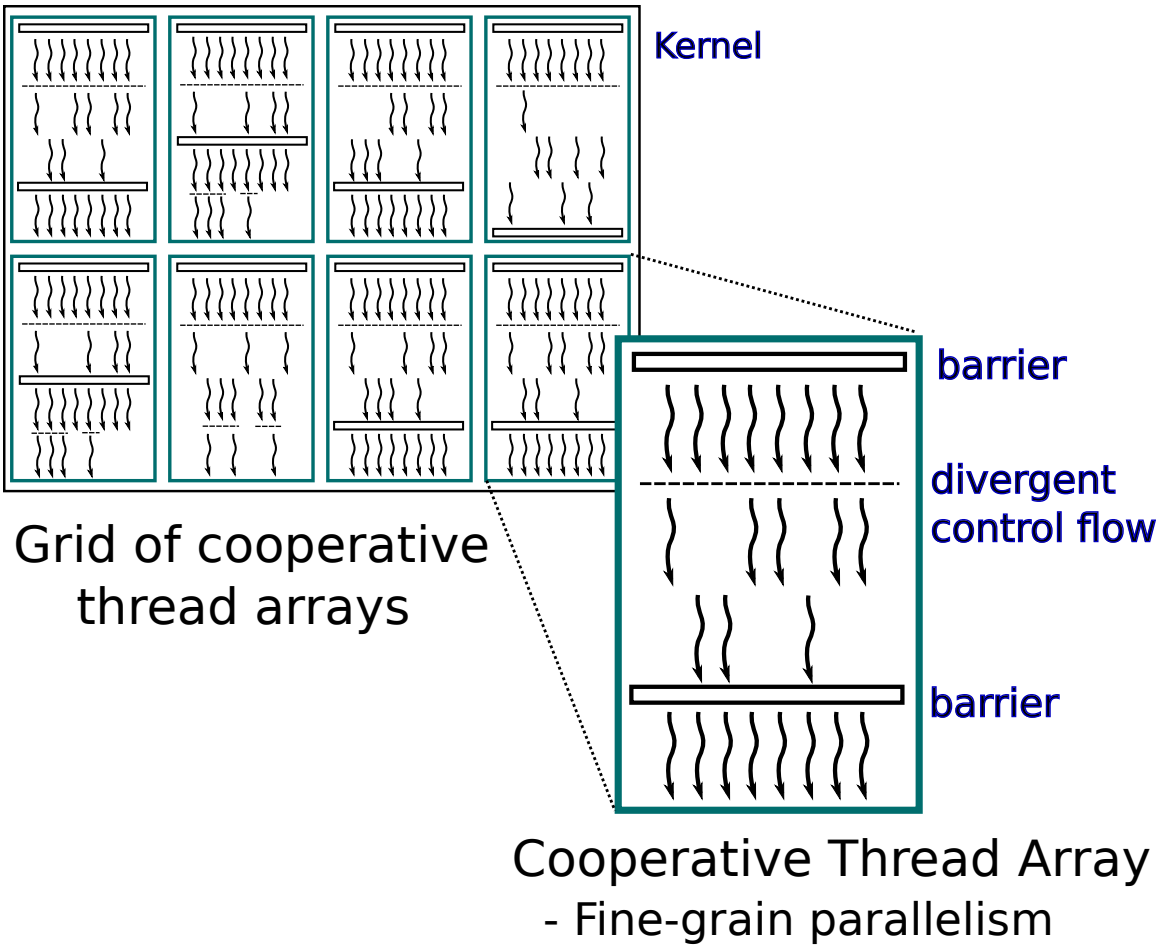

Figure 26: PTX thread hierarchy.

from sequential consistency at the thread level to weak consistency with synchronization points at the CTA level. Threads within a CTA are assumed to execute in parallel, with an ordering constrained by explicit synchronization points. These synchronization points become problematic when translating to sequential architectures without software context switching: simple loops around CTAs incorrectly evaluate synchronization instructions. Stratton et. al. [138] show that CTAs with synchronization points can be implemented without multithreading using loops around code segments between synchronization points. CTAs also have access to an additional fixed size memory space called shared memory. PTX programs explicitly declare the desired CTA and shared memory size; this is in contrast to the warp size, which is

determined at runtime.

The final level of hierarchy in PTX groups CTAs into kernels. CTAs are not assumed to execute in parallel in a kernel (although they can), which changes the memory consistency model to weak consistency without synchronization. Synchronization at this level can be achieved by launching a kernel multiple times, but PTX intentionally provides no support for controlling the order of execution of CTAs in a kernel. Communication among CTAs in a kernel is only possible through shared read-only data structures in main memory and a set of unordered atomic update operations to main memory.

Collectively, these abstractions allow PTX programs to express an arbitrary amount of data parallelism in an application: the current implementation limits the maximum number of threads to 2^{41} [110]. The requirements of weak consistency with no synchronization among CTAs in a kernel may seem overly strict from a programming perspective. However, from a hardware perspective, they allow multiprocessors to be designed with non-coherent caches, independent memory controllers, wide SIMD units, and hide memory latency with fine grained temporal multithreading: they allow future architectures to scale concurrency rather than frequency. The next section shows how a dynamic compiler can harness the abstractions in PTX programs to generate efficient code for parallel architectures other than GPUs.

7.1.5 Translating the PTX Model

The goal of Ocelot is to provide a just-in-time compiler framework for mapping the PTX BSP model onto a variety of many-core processor architectures. The existence of such a framework allows for the same high level representation of a program to be executed efficiently on different processors in a heterogeneous system. It also enables the development and evaluation of a new class of parallel-aware dynamic compiler optimizations that operate directly on an IR with explicit notions of threads,

synchronization operations, and communication operations. Rather than focusing on a single core and relying on a runtime layer to handle processors with multiple cores, the compiler can treat a many-core processor as a single device: it is responsible for generating code that fully utilizes all of the core and memory resources in a processor.

This topic has previously been explored from three complementary perspectives: 1) a static compiler from CUDA to multi-core x86 described in Stratton et al. [138] and extended by the same authors in [137], and 2) a dynamic compiler from PTX to Cell by Diamos et al. [44], and 3) a characterization of the dynamic behavior of PTX workloads by Kerr et al. [84]. From this body of work, the following insights influenced the design of the Ocelot dynamic compiler.

- From MCUDA: PTX threads within the same CTA can be compressed into a series of loops between barriers using *thread-fusion* to reduce the number of threads to match the number of cores in a processor.

- From the PTX to Cell JIT: Performing the compilation immediately before a kernel is executed allows the number and configuration of threads to be used to optimize the generated code.

- From PTX Workload Characterization: Dynamic program behavior such as branch divergence, inter-thread data-flow, and activity factor can significantly influence the most efficient mapping from PTX to a particular machine.

7.1.6 Thread Fusion

Mapping CTAs in a PTX program onto set of parallel processors is a relatively simple problem because the execution model semantics allow CTAs to be executed in any order. A straightforward approach can simply iterate over the set of CTAs in a kernel and execute them one at a time. Threads within a CTA present a different problem because they are allowed to synchronize via a local barrier operation, and

therefore must be in-flight at the same time. In MCUDA, Stratton et al. suggest that this problem could be addressed by beginning with a single loop over all threads and traversing the AST to apply "deep thread fusion" at barriers to partition the program into several smaller loops. Processing the loops one at a time would enforce the semantics of the barrier while retaining a single-thread of execution. Finally, "universal" or "selective" replication could be used to allocate thread-local storage for variables that are alive across barriers.

MCUDA works at the CUDA source and AST level, whereas Ocelot works at the PTX and CFG level. However, Ocelot's approach applies the same concept of fusing PTX threads into a series of loops that do not violate the PTX barrier semantics and replicating thread local data. It differs substaintially in that explicit thread scheduling logic is inserted at the entry points to regions of the program, whereas MCUDA simply loops over all threads, which are represented by an induction variable.

Just-In-Time Compilation With the ability of the compiler to i) fuse threads together [137], ii) redefine which threads are mapped to the same SIMD units [51], iii) re-schedule code to trade off cache misses and register spills [98], and iv) migrate code across heterogeneous targets [44], recompiling an application with detailed knowledge of the system and a dynamic execution profile can result in significant performance and portability gains.

Several challenges typically faced by dynamic binary translation systems were simplified by the CUDA programming model and exploited in the design of Ocelot. Most significantly, 1) kernels are typically executed by thousands or millions of threads, making it significantly easier to justify spending time optimizing kernels, which are likely to be the equivalent of hot paths in serial programs; 2) the self-contained nature of CUDA kernels allows code for any kernel to be translated or optimized in parallel with the execution of any other kernel, without the need for concerns about

thread-safety; 3) kernel code and data segments in PTX are kept distinct and are registered explicitly with the CUDA Runtime before execution, precluding any need to distinguish between code and data by translating on-the-fly.

Profile-Aware Compilation Effective dynamic compilation requires low overhead and accurate predictions of application performance to apply optimizations intelligently. Kerr et al. have recently identified several metrics that can be used to characterize the behavior of PTX applications [84]. Example metrics include the amount of SIMD and MIMD parallelism in an application, control flow divergence, memory access patterns, and inter-thread data sharing. Bakhoda et al. [10] and Collange et al. [33] take a more architecture-centric approach by showing the impact of caches, interconnect, and pipeline organization on specific workloads. Taken together, this body of work provides basis for identifying memory access patterns, control flow divergence, and data sharing among threads as key determinants of performance in PTX programs. Ocelot's many-core backend focuses on efficiently handling these key areas.

In addition to revisiting the insights provided by previous work, the design of Ocelot exposed several other problems not addressed in prior work, most significantly 1) on-chip memory pressure, 2) context-switch overhead, and 3) variable CTA execution time.

7.2 Thread Fusion

Mapping a Bulk-Synchronous Parallel (BSP) application to a processor typically involves a transformation that fits the smallest unit of work (a thread in PTX) to the hardware components. This dissertation refers to this process as **thread serialization**, where the complete set of threads is expressed as an iteration space that can be either unrolled completely and executed by individual threads, or partitioned into sections that can be executed sequentially by individual threads. Thread serialization

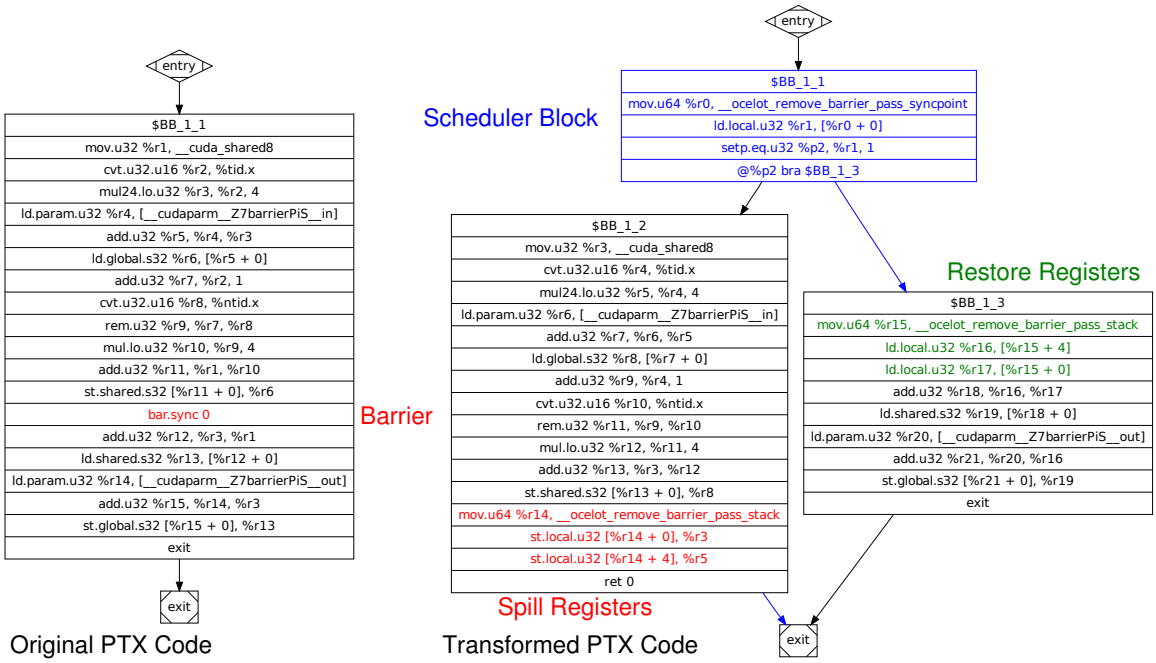

Figure 27: Example of PTX Barrier Conversion. Blue edges are branch targets and black edges are fall-through targets. The left CFG is split at the barrier, all live registers are spilled before the split point, a scheduler block is inserted to direct threads to the resume point, and all live registers are restored at the resume point.

performs transformations that are similar to common loop nest optimizations. The major difference is that the program begins in a completely unrolled form with no loop-carried dependences, making it significantly easier to optimize than traditional loop parallelization approaches that must recover the iteration space first.

7.2.1 PTX Thread Serialization

The semantics of the PTX barrier instruction state that all threads execute up to the barrier before any thread executes beyond the barrier. In order to handle this case, each kernel is broken into sub-kernels beginning at either the program entry point or a barrier, and ending at either a barrier or the program exit point. The solution is to iterate over each sub-kernel one thread at a time to ensure that the semantics of a barrier are retained. However, it is still necessary to handle registers that are live across the barrier because they represent thread-local state that would otherwise be lost during a context switch.

Live registers are maintained by creating a register spill area in local memory for each thread. For each kernel exit point ending in a barrier, all live registers are saved to the spill area before exiting the kernel. For every kernel entry point beginning with a barrier, code is added that restores live registers from the spill area. The definition of local memory ensures that the spill area will be private for each thread, so this transformation can be applied directly at the PTX level.

Figure 27 shows an instructive example of this process. The left kernel contains a single basic block with a barrier in the middle. The right figure shows the program control flow graph after removing barriers. The immediate successor of the entry block decides whether to start from the original kernel entry point or the barrier resume point. The left successor of this block is the original kernel entry point and the right block is the barrier resume point. Note that two live registers are saved at the end of the left-most node. They are restored in the rightmost block. During execution, all threads will first execute the leftmost block then they will execute the rightmost block and exit the kernel.

For a series of barriers, multiple resume points will be created. After translation, the LLVM optimizer is used to convert the chain of entry blocks into a single block with an indirect jump. This approach is logically equivalent to deep-thread-fusion as described by Stratton et al. [137], although it works with the PTX control-flow-graph rather than the CUDA AST.

7.3 Subkernel Formation

This approach of compiling entire kernels on-demand as they are executed for the first time is based on the assumption that kernels will have good code coverage; that is, that most of the code in the kernel will be executed frequently enough to justify the overheads of compilation. These characteristics paint an accurate picture of Harmony and CUDA kernels that exist today because of the nascence of these execution models:

most kernels correspond to the inner loops from high performance applications. The matrix multiplication, AES encryption, and sum-of-absolute-difference kernels used throughout the experimental section of this dissertation are examples of this trend.

However, historical examples have shown that as programming models shift from supporting high performance applications that deal with special cases to high productivity applications that deal with general cases, code size increases, code coverage decreases, and application performance becomes more sensitive to call intensive rather than loop intensive regions of the code [32]. Emerging PTX backends for LLVM [127], .NET [145], and Python [24] along with template meta-programming libraries for CUDA are a step in this direction.

Some kernels are beginning to experience this issue: translating merge and sorting kernels from commonly used template libraries [66] takes longer than the kernel execution time on existing GPUs (the NVIDIA 480GTX for example). Existing solutions for this problem focus on switching control back and forth between the dynamic compilation system and the kernel such that new sections of the program are compiled as they are encountered [11, 18, 21, 29]. This works well for single threaded programs, but suffers from resource contention issues in multi-threaded environments [17]. Some solutions have been proposed for systems with a limited number of threads and cores [17], but the explicitly parallel representation of PTX would typically require thousands or millions of threads to compete for access to the dynamic compiler.

This section introduces a potential solution to this problem with the concept of a **sub-kernel**. Complete kernels are decomposed into sections that are referred to as sub-kernels at compile time before they are executed. During execution threads are bulk-scheduled; they are batch-launched through sub-kernels and gated as they leave sub-kernels. This creates natural boundaries to perform lazy compilation as a batch of threads attempts to enter a sub-kernel that has not yet been compiled. It also presents a pool of threads that can be used to perform the compilation in parallel.

An overview of the implementation of sub-kernel formation in Ocelot is described as a case study in Chapter 8.

7.4 The Ocelot Dynamic Compiler

Harmony relies on either the existence of multiple binaries for each kernel or a dynamic compiler that can generate code for new processors on demand. The lack of a commercially available dynamic compilation environment[1] lead to the development of Ocelot, a dynamic compilation framework that provides back-end targets for the NVIDIA PTX virtual ISA [37]. Ocelot currently supports NVIDIA GPUs and multi-core CPUs and has been validated on over 130 applications. Using Ocelot as a dynamic compiler, Harmony applications can be written as a collection of PTX kernels, where the existing NVIDIA compilation chain is leveraged to compile CUDA kernels into PTX binaries. This section details the design of the Ocelot compiler and how the PTX execution model that was originally intended to be executed on NVIDIA GPUs can be mapped to multi-core CPUs, enabling Harmony applications to execute on systems with NVIDIA GPUs and multi-core CPUs.

7.4.1 Dynamic Optimization

The CPU back-end currently leverages the LLVM framework for optimization passes that are applied after the parallel PTX execution model is transformed into the sequential LLVM execution model. However, it currently does not take advantage of the SIMD parallelism inherent in PTX to generate vector instructions for use in CPU SSE/Altivec units. This could provide up to a 4x speedup over the current implementation and up to an 8x improvement when AVX is introduced. Finally, Ocelot includes an optimization framework for PTX that supports state of the art compiler optimizations that can consider the synchronization primitives, communication

[1] OpenCL had not been released at the time that the Ocelot project was started.

between threads, and the explicit memory hierarchy exposed in PTX.

7.5 Alternative Execution Models

The previous sections have shown that the PTX model can be efficiently compiled and executed on several different classes of processing elements.

The right choice depends on the structure of the computation being performed and the organization of the system at a given level of granularity. As the level of heterogeneity increases, execution models such as Harmony may become a better match, and a complete programming model for a multi-level system may be composed of a different execution model that matches the degree of structure or heterogeneity at each level.

CHAPTER VIII

CASE STUDIES

Up to this point, this dissertation has covered the abstractions in the Harmony execution model, an execution model for individual kernels, optimizations that can be applied each model, and possible mappings of the model onto several systems at different granularities. This chapter includes a series of six detailed evaluations of these topics presented in the form of case studies. The first case study examines the limits of kernel level parallelism in several Harmony and CUDA applications. The next study covers the design and performance evaluation of an implementation of the Harmony runtime targeting systems with directly connected accelerator boards supporting memory renaming and speculative kernel execution. Another study focuses on the performance of PTX kernels that are compiled for x86 CPUs using Ocelot. The fourth case study extends this work by introducing a concept of a subkernel that can ease the process of dynamically translating between explicitly parallel kernel execution models. The fifth study explores the problem of building a predictive performance model for kernel execution time using empirical techniques. The final study describes the design of a language frontend and high level compiler for Harmony.

Application	Description	Problem Size	Control Flow	Model
AES	Encrypts and decrypts a large document using 256-bit AES	3.2 MB Text File	For Loops	Harmony
MonteCarlo	Gaussian Quadrature estimates the area under a normal function	1 Precision value	While Loops	Harmony
MatrixMultiply	Dense matrix multiplication using subblocks	4096x4096 matrices	Nested Loops	Harmony
CapModel3	Risk analysis for adding a new asset to an existing loan portfolio	1000000 Samples	Nested Loops	Harmony
Random	A regression test for Harmony that constructs a CFG of simple kernels with random edges subject to a completion constraint	10 Variables 100 Average Iterations	Random Structure	Harmony
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
MRI-FHD	Computation of an image-specific matrix FHd, used in a 3D MRI reconstruction algorithm in non-Cartesian space.	450KB Image	For Loops	CUDA
CP	Computes the coulombic potential at each grid point over on plane in a 3D grid in which point charges have been randomly distributed.	40000 Atoms in a 512x512 grid	For Loops	CUDA
SAD	Sum of absolute differences kernel, used in MPEG video encoders.	50KB image	None	CUDA
TPACF	Measures the probability of finding an astronomical body at a given angular distance from another astronomical body.	100 Random Numbers 4096 Points	None	CUDA
PNS	Implements a generic algorithm for Petri net simulation.	2000x2000 matrix	For Loops	CUDA
RPES	Calculates 2-electron repulsion integrals which represent the Coulomb interaction between electrons in molecules.	20000 molecules	For Loops	CUDA

Table 1: Application Characteristics

8.1 The Limits of Kernel Level Parallelism

8.1.1 Kernel Level Parallelism in Harmony Applications

This study develops a theoretical formulation of the upper bound of KLP in Harmony and CUDA applications. It shows how this upper bound is impacted by speculation and renaming for the benchmark applications in Table 1. The Harmony applications were written from scratch for this thesis and the CUDA applications were taken from the UIUC Parboil benchmark suite [72].

At a high level, KLP should express the amount of parallelism within a Harmony or CUDA application in the same way that ILP expresses the amount of instruction level parallelism within a single threaded application. KLP is difficult to formulate exactly as different kernels typically have different, data-dependent execution times as shown in Diamos et al. [47] and Luk et al. [95]. Furthermore, these execution

Application	Kernels	KLP	MIMD	SIMD
CP	10	9.85	256	128
MRI-Q	4	3.91	97.5	320
MRI-FHD	7	6.96	110.57	292.57
SAD	3	2.6	594	70.28
TPACF	1	1.0	156.63	206.11
PNS	112	111.03	17.99	248.88
RPES	71	70.42	64757	40.5799

Table 2: Parallelism in CUDA Applications

times can be dependent on input data and are typically strongly dependent on processor architecture. They can even include non-deterministic components when run on systems with stateful hardware units like caches, variable performance characteristics caused by DVFS, and software artifacts such as operating system scheduling or dynamic binary translation and optimization.

With these concerns in mind, *kernel level parallelism is defined for an application on a heterogeneous system as the speedup of a parallel execution on a system with an infinite number of accelerators over a sequential execution on the same system where each kernel is run on the accelerator that gives the lowest execution time.* In order to account for possible non-determinism in the execution time of a kernel, the average execution time from the accelerator with the lowest such average time is used.

For Harmony applications, KLP is computed by analyzing traces of the kernels and control decisions launched by an application as it executed. These traces express the average kernel execution time and the input and output variables of each kernel. For CUDA applications, Ocelot [45] was used to instrument all load and store instructions from every kernel in an application. A set of all memory locations written by kernels was maintained along with the id of the last kernel to write to that location. If a kernel ever loaded a value that was stored by a previous kernel, a dependency was created between the two kernels. This information was used to create a dependency graph for the entire application. Ocelot did not have the ability to identify control

decisions in CUDA applications because it could only analyze PTX kernels, not the complete application, so the best case KLP assuming that all control decisions could be removed via perfect speculation is reported in Table 2. The average MIMD and SIMD parallelism as defined in Kerr et al. [84] within each kernel is also presented for comparison.

MIMD parallelism is computed as the speed up of a GPU with an infinite number of multiprocessors over a GPU with a single multiprocessor, ignoring memory bandwidth and latency constraints. It is computed by assuming that each instruction takes a single cycle to complete and dividing the total number of dynamic instructions by the dynamic instruction count (D_i) of the longest running CTA in a kernel as shown in Equation 1. It is averaged over all kernels as in Equation 2. **SIMD parallelism**, on the other hand, is computed as the average activity factor (A_f) of a CTA, multiplied by the number of threads in the CTA, and weighted by the number of dynamic instructions in the CTA as shown in Equation 3. It is averaged over all CTAs in a kernel as in Equation 4.

$$MIMD_{kernel} = \frac{\sum_{i=1}^{ctas} D_i}{\max_{i=1}^{ctas}(D_i)} \tag{1}$$

$$MIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * MIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i} \tag{2}$$

$$SIMD_{kernel} = \frac{\sum_{i=1}^{ctas} A_f * D_i}{\sum_{i=1}^{ctas} D_i} \tag{3}$$

$$SIMD_{application} = \frac{\sum_{i=1}^{kernels} D_i * SIMD_{kernel\_i}}{\sum_{i=1}^{kernels} D_i} \tag{4}$$

Figure 28 shows the computed upper bound on KLP for all of the Harmony applications in our test suite. In this figure, speculative depth refers to the maximum number of control decisions that can be outstanding at a time. These results show a

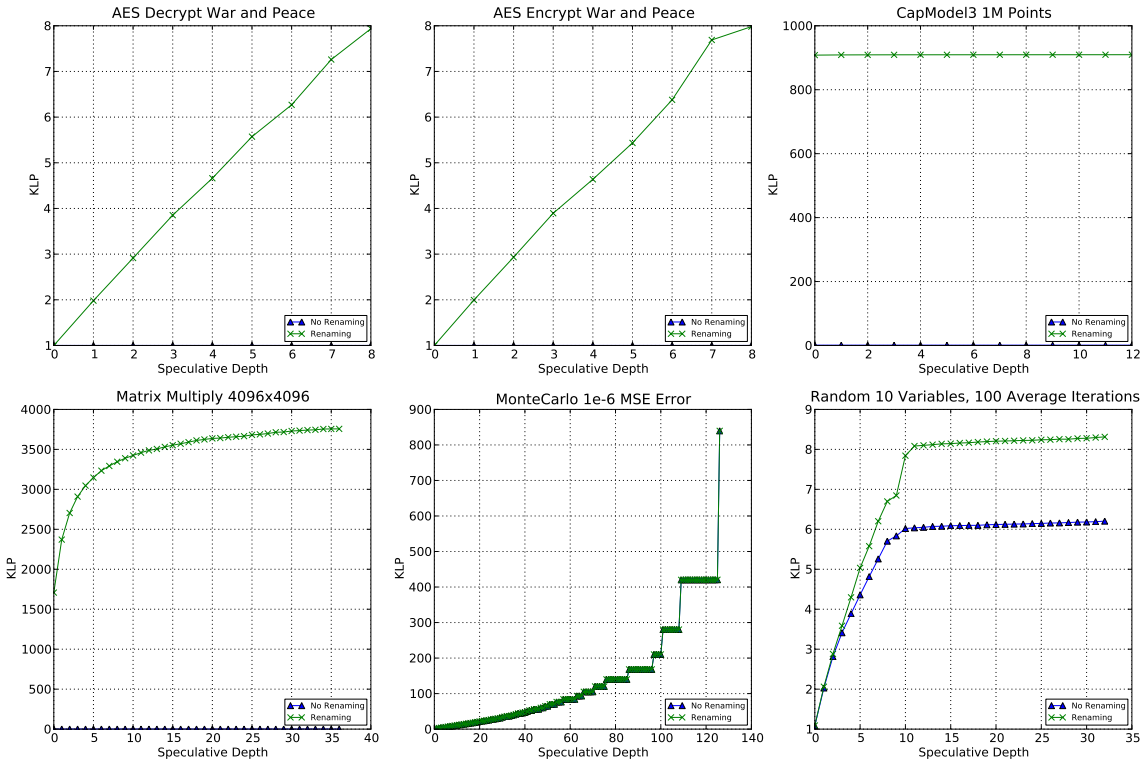

Figure 28: Upper bound on kernel-level-parallelism in Harmony applications. The blue line represents KLP when variable renaming is not allowed. The green line represents KLP with renaming enabled.

significant amount of KLP within all applications tested. For all of the applications except Monte Carlo, renaming provides the most significant boost to KLP and indeed most applications without renaming do not show any improvement from speculation. Monte Carlo stands out because it explicitly uses different variables for the input seed and output result of each Monte Carlo simulation; this demonstrates that it is possible for the programmer to do the equivalent of renaming. However, once renaming has been enabled, being able to remove control decisions via speculation greatly improves KLP. Over all of the applications, it extends the upper bound of KLP by an average of 3.6x over renaming alone. The KLP saturates around a speculative depth of about 10 except for Monte Carlo which continues to scale up to a speculative depth of 133.

For the CUDA applications, KLP is comparable to that of the Harmony applications in all cases except for SAD, TPACF and MRI-Q, which simply do not launch a

CPU	Intel i920 Quad-Core 2.66Ghz
Memory	8GB DDR-1333 DRAM
CPU Compiler	GCC-4.4.0
GPU Compiler	NVCC-2.3
OS	64-bit Ubuntu 9.10

Table 3: Test System

significant number of kernels. It is possible that increasing the data set size for these benchmarks would improve their KLP. It is also possible that the kernels in these applications would have to be split to expose additional KLP, which would limit the potential benefits of speculation. Kerr et al. [84] show that this is not the common case for CUDA applications, which typically have tens to hundreds of kernels, but it still represents a problem that will have to be addressed in future work in order to apply kernel level speculation to certain CUDA applications.

8.2 The Design of The Harmony Runtime

This case study describes the implementation of the Harmony runtime used throughout this dissertation. The runtime itself is broken out into components that perform different functions such as kernel dependency tracking, performance prediction, or scheduling. This is followed by a series of experiments that measure the overheads of each component, and the performance scaling of Harmony applications on a system composed of directly connected accelerator boards.

8.2.1 The Runtime Components

The high level structure of the runtime is broken out into a program binary interface, a kernel fetch unit, a kernel dependency graph, a memory manager, a kernel scheduler, a device interface, and a performance predictor. A window of kernels is read from the program binary by the kernel fetch unit, some of which are flagged as speculative. These are added to the kernel dependency graph, which tracks all kernels and their data dependences. The memory manager maintains the set of all variables in

```
message KernelControlFlowGraph
{
    required string     name      = 1; // Program Name
    repeated BasicBlock blocks    = 2; // List of basic blocks in the program
    repeated Variable   variables = 3; // Global variables in the program
    required uint32     entry     = 4; // Id of the program entry block
    required uint32     exit      = 5; // Id of the program exit block
}

message Variable
{
    required uint32 name = 1; // Unique identifier of the variable
    required bytes  data = 2; // Initial data of the variable
}

enum AccessMode
{
    In    = 1; // Operand is read-only
    Out   = 2; // Operand is destructive write-only
    InOut = 3; // Operand is read-write
}

 message Operand
 {
     required AccessMode mode     = 1; // How the kernel can access this
     required uint32     variable = 2; // Associated variable
 }

 enum KernelType
 {
     ComputeKernel       = 1; // Conumes inputs, produces outputs
     ControlDecision     = 2; // Changes control flow, multiple targets
     UnconditionalBranch = 3; // No operands, jumps to the first target
     GetSize             = 4; // Get the size of a variable in bytes
     Resize              = 5; // Change the size of a variable in bytes
     Exit                = 6; // No operands, ends the program
 }
 message Kernel
 {
     required string     name     = 1; // Name for debugging
     required string     ptx      = 2; // PTX assembly for the kernel
     required KernelType type     = 3; // Compute kernel or Control Decision
     repeated Operand    operands = 4; // Kernel operands
     repeated uint32     targets  = 5; // Target block ids if this is a Control
 }

 message BasicBlock
 {
     required uint32 id       = 1; // Unique ID of the block
     repeated Kernel kernels  = 2; // List of kernels in the block
     required Kernel control  = 3; // Determines the next block to execute
 }
```

Figure 29: The google protocol buffer specification of the Harmony binary format. Messages are encoded and serialized into a binary file.

the system, renamed copies, and their mapping to memory regions. The performance predictor is used to determine the expected execution time of kernels on different processors in the system as well as model costs of dynamic compilation and variable copies between memory regions. The kernel scheduler examines the kernel dependency graph and the performance predictor to assign kernels to processing elements at specific times. The processor interface is used as the mechanism for compiling and launching kernels. It is connected directly to the Ocelot dynamic compiler.

The Program Representation Harmony programs are stored as a serialized control flow graph of PTX kernels. Kernels are packed into basic block units, and the entire program is represented as a list of these units along with links between blocks representing control dependences between blocks. Figure 29 shows the google protocol buffer specification [149] of the file format. Harmony will open a file in this format, map it into memory, and lazily load basic blocks as they are requested by the kernel fetch unit. Note that the PTX assembly code for each kernel is embedded in the actual kernel object. Actually loading these binaries from disk can account for a significant portion of the compilation time, and it is advantageous to also defer the loading of the PTX assembly until the runtime actually requests access to the kernel, indicating that it is likely to be executed.

The Kernel Fetch Unit The kernel fetch unit drives the execution of a Harmony program, which consists completely of kernels and no additional native code. It walks the kernel CFG in program order to create a sequence or trace of kernels that should be executed next. Kernels are fetched one basic block at a time and a software branch predictor is used to speculatively fetch kernels across basic block boundaries. Once a kernel is fetched, it is placed into an outstanding state that indicates that it is queued for execution, but has not yet completed. A threshold is placed on both the number of kernels that can be fetched speculatively and the total number of outstanding kernels so that only enough kernels to saturate the machine are queued up for execution.

The Memory Manager The memory manager tracks all variables in a Harmony program, their mapping into memory regions, and the sequence of operations that are scheduled to modify each variable. When kernels are fetched, their parameter list is examined and read-mode operands are mapped to the current version of the corresponding variable. Write operands are also scheduled on the current version of the corresponding variable assuming that either 1) there are no outstanding read or

write operations scheduled on that variable, or 2) there is no space available to rename the variable. Otherwise, the variable will be renamed and the kernel completion, removes scheduled operations, and destroys renamed variables that will write to the newly created copy. The memory manager also tracks kernel are no longer referenced by any kernels.

The Kernel Dependency Graph After kernels have been fetched and their variable operands have been bound, they are passed to the kernel dependency graph. The dependency graph maintains a list of kernels that are waiting on other kernels to complete before they can begin executing. An initial list of dependences is provided by the Memory Manager when the kernel is passed to the dependency graph. As kernels complete, dependent kernels are updated, and are removed from the dependency graph if they have no more dependences. The dependency graph can also be traversed by the kernel scheduler to schedule chains of dependent kernels rather than performing scheduling once kernel dependences have been resolved.

The Performance Predictor The performance predictor maintains a running prediction of the execution time of each kernel from the time that it is fetched until the time that it completes. As more information is made available, for example, 1) which variables it uses, 2) information about its PTX assembly code, and 3) a history of similar kernels or previous executions of the same kernel, the prediction is refined. The performance predictor allows the other components to query for a prediction of the execution time of any kernel on any processor in the system. This prediction includes an absolute value as well as a confidence interval for the prediction. The other components may choose to use the prediction as it is or query again after a higher confidence prediction is obtained. The predictor works by recording a window of kernel execution times and other parameters such as the machine parameters of the processor it executed on, the size and distribution of input parameter variables, and

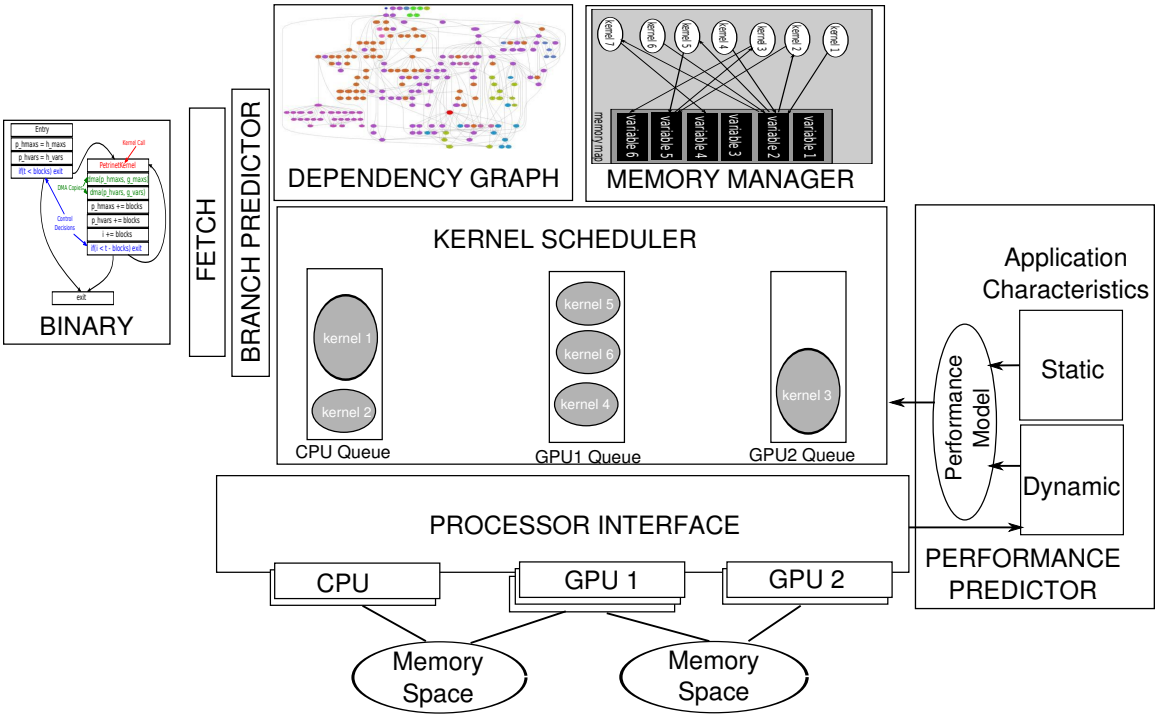

Figure 30: An overview of the implementation of the Harmony Runtime.

other instrumentation data. These parameters are fed into a polynomial regression model that creates a function relating these parameters to the expected execution time of the kernel.

The Kernel Scheduler The kernel scheduler examines the kernel dependency graph to get the set of outstanding kernels that have been fetched but not yet scheduled. The commonly-used list scheduling algorithm [56] is used to determine which processing element to schedule the kernel on. However, this is only one function of the scheduler. It also sorts the kernel graph by criticality, such that more critical kernels are scheduled first. Criticality in this context is defined as the sum of execution times of kernels that directly or indirectly depend on the current kernel. Also, it will re-compute parts of the schedule under certain circumstances:

- Misspeculation: A misspeculation will result in several kernels being removed

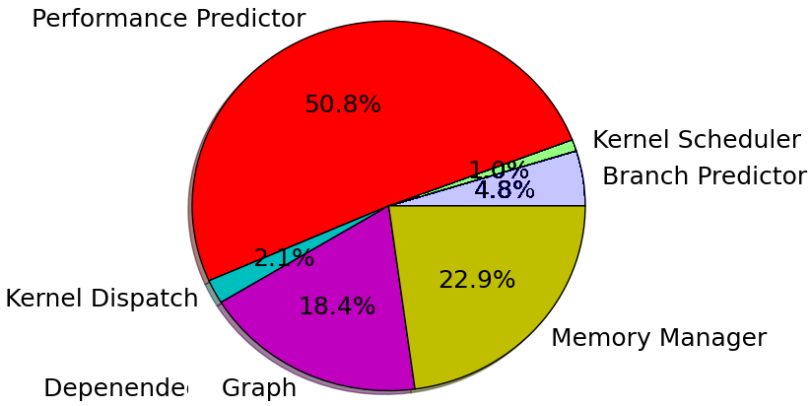

Figure 31: The breakdown of time spent in each component of the runtime.

from the scheduling lists. The entire schedule will be recomputed if this happens.

- Load imbalance: Load imbalance is detected when a scheduling list becomes empty and lists for other processors have excess kernels. In this case, kernels will be removed from the longest list and rescheduled until either there is no change in the schedule, or the all of the lists are no longer empty.

The Processor Interface The processor interface provides a single Application Programming Interface (API) for managing processing elements from the Harmony runtime. This interface connects directly to the Ocelot dynamic compiler for managing devices. The Ocelot interfaces for loading PTX binaries, applying different levels of optimization, allocating/copying/freeing memory, selecting devices, and asynchronously launching kernels are used directly. The major limitation of this interface is the lack of an ability to cancel kernels that have begun executing.

8.2.2 Performance Evaluation

This section covers an empirical evaluation of the Harmony runtime running on a highly heterogeneous system. The characteristics of our test system are given in Table 3. The first experiment quantifies the performance overhead of each module

in the runtime. The next experiment compares measured scaling to our KLP model, then present the base case execution time of each Harmony application using multiple system configurations, and conclude with the execution time of the complete system with and without speculation.

Runtime Overheads We used on-chip performance counters to measure the overhead of launching a kernel and, on average, these modules introduce an overhead of 38,724 CPU cycles for each kernel. Figure 31 shows the average breakdown of these cycles into modules. As can be seen, most of the time is spent in the Dependency Graph, the Memory Manager, and the Performance Predictor. Across the applications tested, the runtime overheads are less than 1% of the total execution time indicating that the kernels are relatively coarse-grained compared to the operations performed by the runtime.

KLP Comparison The KLP metric presented in Section 8.1 represents an upper bound on the parallel scaling of a given application. In order to evaluate the effectiveness of our implementation of speculation in relation to the KLP ideal, we first measure the execution time of each application without speculation. The KLP metric for each speculative depth is then normalized to the measured non-speculative time, eg. an application with an execution time of 10s and a KLP of 2 at depth 1 would be predicted to finish in 5s on a machine with at least two cores. For this experiment, we use only CPU cores for both the KLP metric as well as the measured execution time so that scaling trends are easily visible.

Figure 32 shows the measured and KLP predicted execution times. The MonteCalo, CapModel3, and MatrixMultiply follow the prediction closely. Random deviates the most, experiencing no speedup even though the KLP metric predicts a speedup of up to 4x. This can be explained by examining the structure of the application, where kernels only contain several instructions each; the execution time is

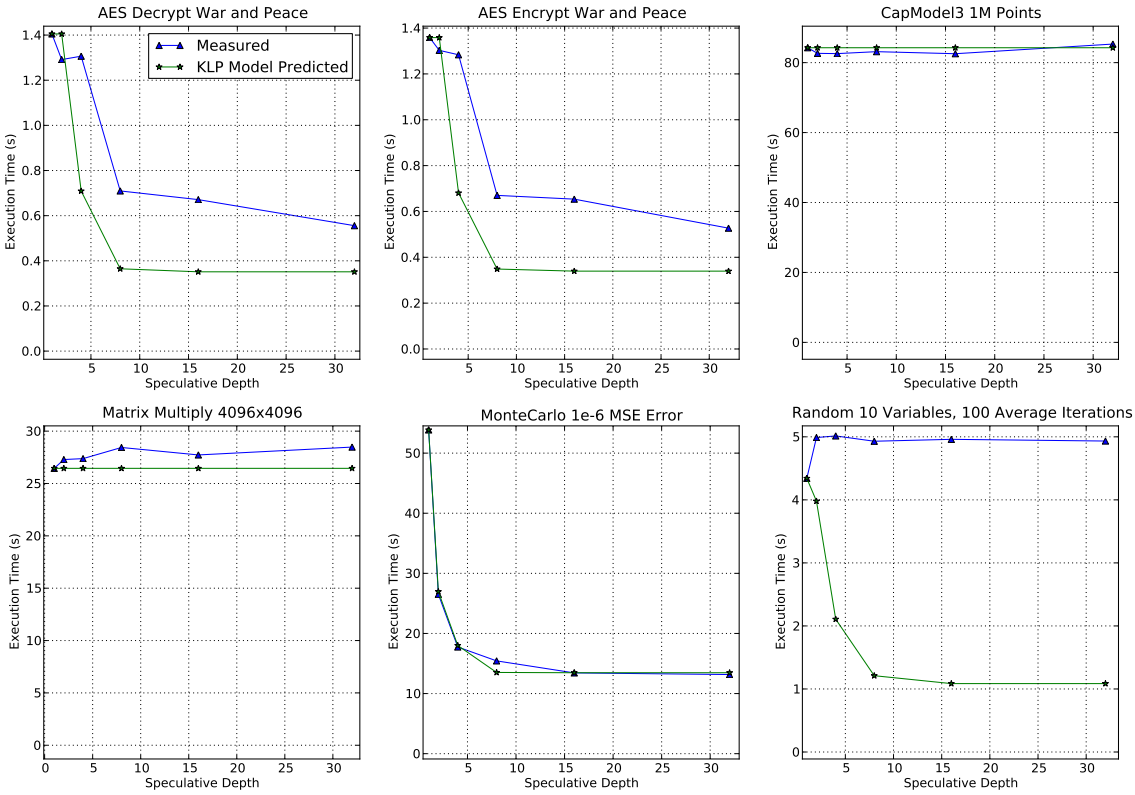

Figure 32: KLP Predicted vs Measured Scaling (4 CPUs)

dominated by runtime overheads, which are serialized. The AES application suffers from inaccuracy as well. Profiling the application shows that it spends most of its time in functions doing memory and disk accesses suggesting that it is IO rather than compute bound, limiting its scalability on a multicore system.

Heterogeneous Scaling This experiment establishes a base case for the speedup of each application using Harmony without KLS[1]. As can be seen in Figure 33, on average, the use of all 10 cores in the system provides a 14.8x increase in performance over a single CPU with the MatrixMultiply example seeing the largest improvement at 57.6x. For MatrixMultiply, the complete system achieves 824 Gflops compared to 201 Gflops achieved by a single 8800GTX GPU in prior work [150].

[1]Random is omitted from these results since it does not use GPU kernels.

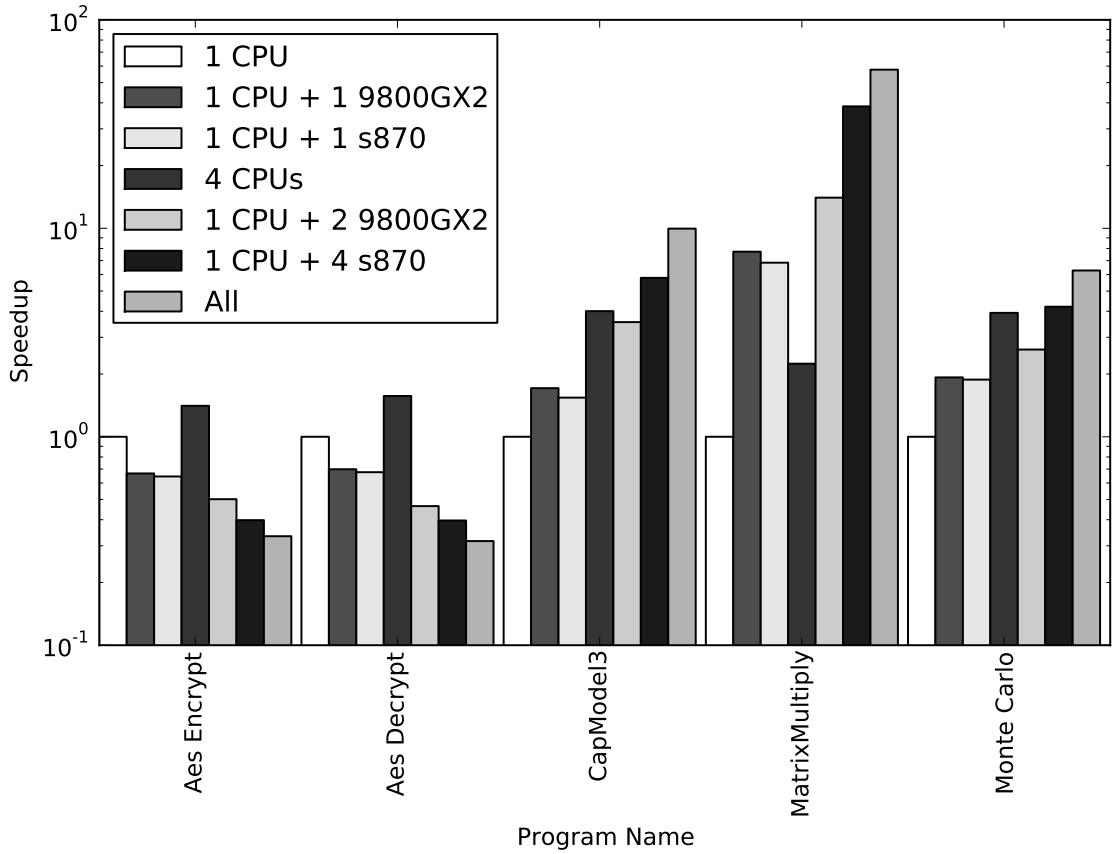

Figure 33: Scaling Without Speculation

Only the AES application experiences a slow-down moving from a single CPU to the entire system. In our GPU implementation, the GPU encrypt and decrypt are at least an order of magnitude slower than their CPU equivalent. This is not a typical result for AES, which others have shown to perform well on GPUs [99], and is likely due to an inefficiency in our implementation. However, it presents an interesting case where a GPU kernel is much slower than a CPU kernel. As these applications only have ten encrypt or decrypt kernels each, running even a single kernel on a GPU core will degrade the performance of the application. Examples like this motivate the refinement of the performance predictor to either try to estimate the execution time on each architecture before actually launching a kernel, or kill extremely long running kernels and restart them on faster architectures.

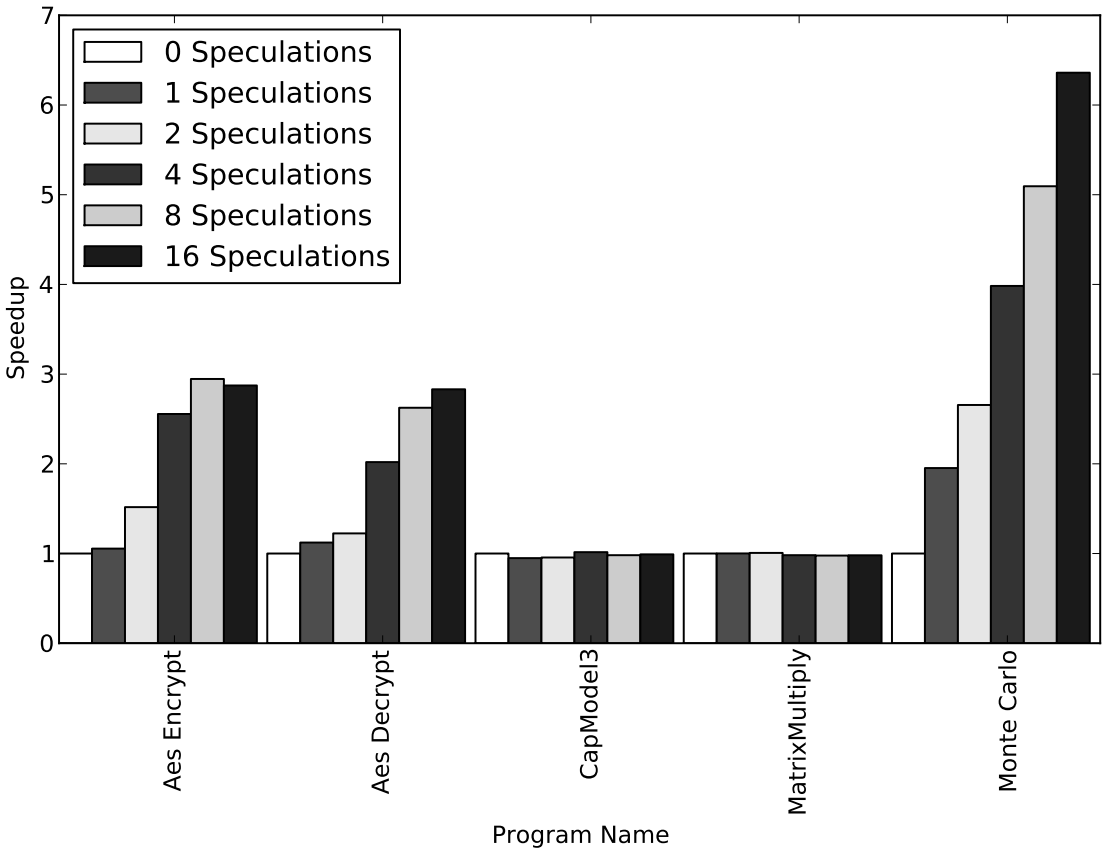

Figure 34: Performance Improvement With Varying Speculative Depth

Additional Scaling Using Speculation The next experiment focuses on the benefits from adding speculation to the base implementation. Figure 34 shows the performance improvement of the entire system moving from the non-speculative implementation to the speculative implementation using various speculative depths. Of the applications that were predicted by the KLP model to benefit from speculation, performance improves by an average of 3.98x. The other two applications, CapModel3 and MatrixMultiply, are not affected at all by the overheads of speculation, experiencing a ±3% change in execution time. These two applications already have enough KLP within each basic block to fully utilize the system, and thus do not require speculation to expose any more. Across all of these applications, adding kernel level speculation either improves performance or does not impact it at all.

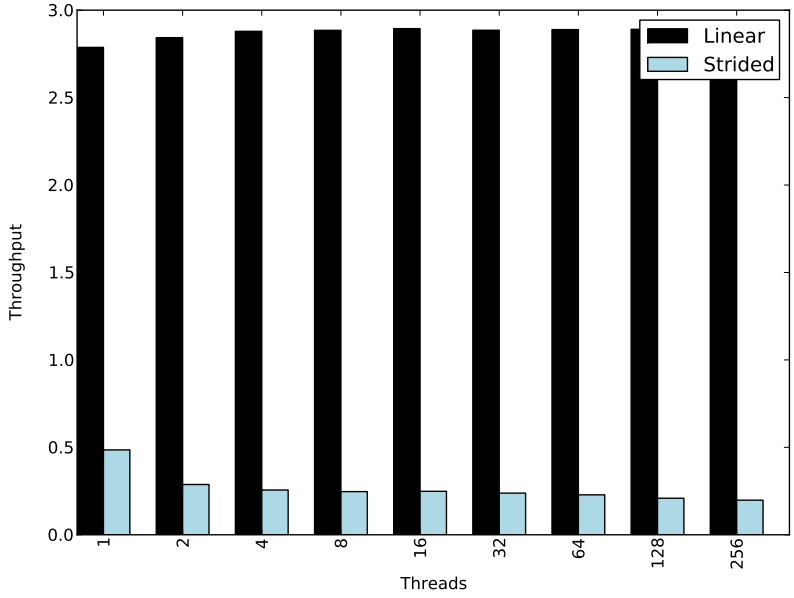

Figure 35: Memory throughput using different access patterns, all executing on a CPU. Note that strided accesses, which result in efficient coalesced accesses on GPUs, are inefficient on the CPU.

8.3 Ocelot: A Prototype Dynamic Kernel Compiler

8.3.0.1 The Ocelot CPU Backend

The next study covers a preliminary analysis of the performance of several CUDA applications when translated to LLVM. Note that this section is intended to merely provide several distinct points of reference of the scaling and throughput possible when translating CUDA applications to x86. All of the experiments in this section use the system configuration given in Table 3. The section begins with a set of experiments exploring the performance limits of Ocelot using a set of microbenchmarks, moving on to a classification of runtime overheads, and ending with a study of the scalability of several full applications using multiple cores.

8.3.1 Microbenchmarks

The first set of experiments focus on several low level PTX benchmarks that were designed to stress various aspects of the system. In order to write and execute PTX programs outside of the NVIDIA compilation chain, which does not accept inlined

assembly, the CUDA Runtime API was extended with two additional functions to allow the execution of arbitrary PTX programs. The function **registerPTXModule** allows inserting strings or files containing PTX kernels at runtime and **getKernelPointer** obtains a function pointer to any registered kernel that can be passed directly to **cudaLaunch**.

```
void registerPTXModule( std::istream& module, const std::string& moduleName );
const char* getKernelPointer( const std::string& kernelName,
     const std::string& moduleName );
```

Using this infrastructure, memory bandwidth, atomic operation throughput, context-switch overhead, instruction throughput, and special function throughput were explored. These measurements were taken from a real system, and thus there is some measurement noise introduced by lack of timer precision, OS interference, dynamic frequency scaling, etc. These results were taken from the same system and include at least 100 samples per metric. The sample mean is presented in the form of bar charts with 95% confidence intervals for each metric.

Memory Bandwidth. The microbenchmark explores the impact of memory traversal patterns on memory bandwidth. This experiment is based on prior work into optimal memory traversal patterns on GPUs [110], which indicates that accesses should be coalesced into multiples of the warp size to achieve maximum memory efficiency. When executing on a GPU, threads in the same warp would execute in lock-step, and accesses by from a group of threads to consecutive memory locations would map to contiguous blocks of data. When translated to a CPU, threads are serialized and coalesced accesses are transformed into strided accesses because their scheduling order is changed. It changes from multiple threads executing the same instruction in lock-step to each thread executing a sequence of instructions and then context switching to the next one. Figure 35 shows the performance impact of this

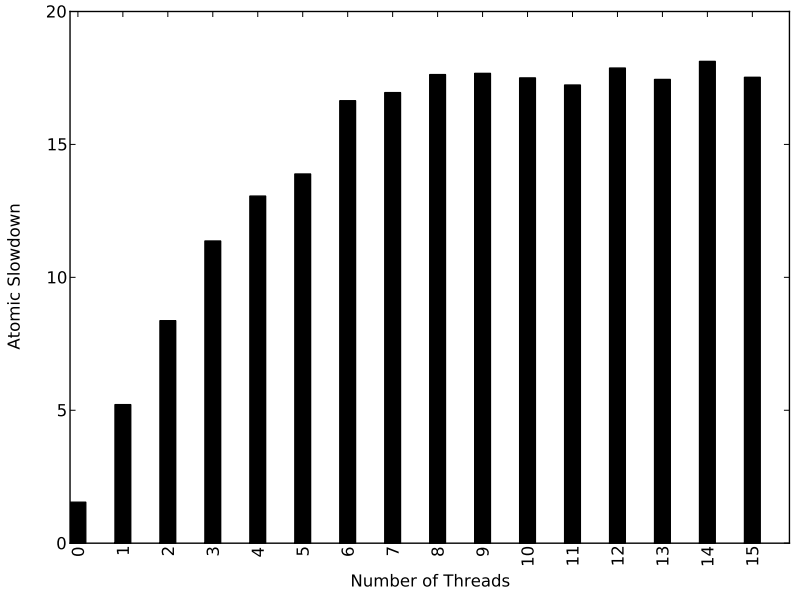

Figure 36: The throughput of atomic operations decreases as the number of host worker threads increases. The overhead of using locks only introduces at worst 20x the overhead over lock-less operations.

change. The linear access pattern represents partitioning a large array into equal contiguous segments and having each thread traverse a single segment linearly. The strided access pattern represents a pattern that would be coalesced on the GPU. It is very significant that the strided access pattern is over 10x slower when translated to the CPU. This indicates that the optimal memory traversal pattern for a CPU is completely different than that for a GPU.

Atomic Operations. The next experiment details the interaction between the number of host worker threads and atomic operation overhead. This experiment uses an unrolled loop consisting of a single atomic increment instruction that always increments the same variable in global memory. Each PTX thread executes a single iteration of the unrolled loop. The loop continues until the counter in global memory reaches a preset threshold. As a basis for comparison, the same program was run using a single thread to increment a single variable in memory until it reached the same threshold without using atomic operations. Figure 36 shows the slowdown of

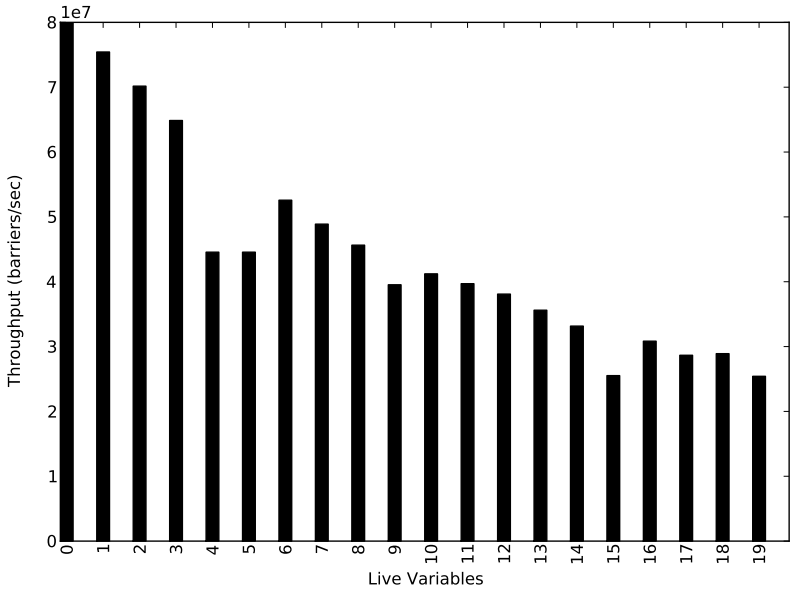

Figure 37: The overheads of context switches are primarily due to spilling live registers. Applications with few live variables are likely to perform well on CPUs.

the atomic increment compared to the single-thread version for different numbers of CPU worker threads. These results suggest that the overhead of atomic operations in Ocelot are not significantly greater than on GPUs.

Context-Switch Overhead. The third experiment explores the overhead of a context-switch when a thread hits a barrier. It consists of an unrolled loop around a barrier, where several variables are initialized before the loop and stored to memory after the loop completes. This ensures that they are all alive across the barrier. In order to isolate the effect of barriers on a single thread, only one thread and one CTA were launched for this benchmark. A thread will hit the barrier, exit into the Ocelot thread scheduler, and be immediately scheduled again.

Figure 37 shows the measured throughput, in terms of number of barriers processed per second. Note that the performance of a barrier decreases as the number of variables increases, indicating that a significant portion of a context-switch is involved in saving and loading a thread's state. In the same way that the number of live registers should be minimized in GPU programs to increase the number of threads that

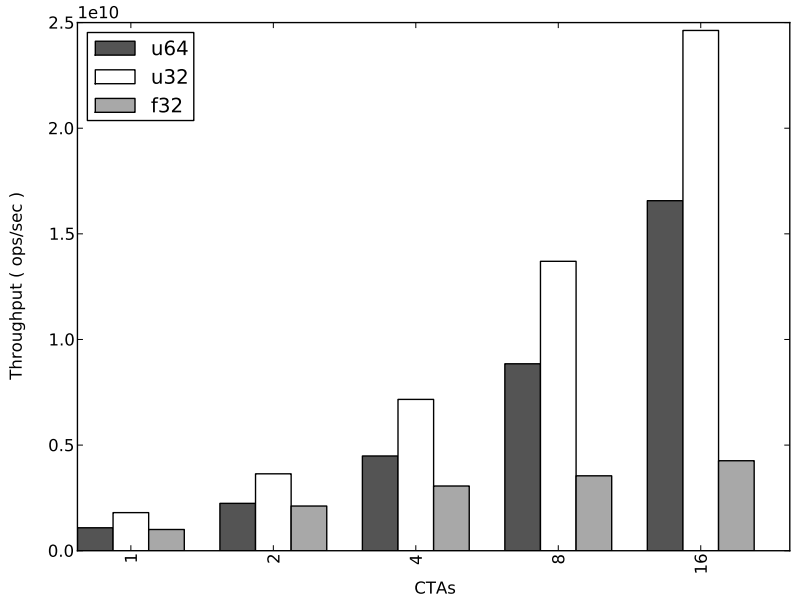

Figure 38: CPUs are more efficient when performing integer operations.

can be active at the same time, programs translated to the CPU should actively try to minimize the number of live registers to avoid excessive context-switch overhead.

Instruction Throughput. The fourth microbenchmark attempts to determine the limits on integer and floating point instruction throughput when translating to a CPU. The benchmark consists of an unrolled loop around a single PTX instruction such that the steady state execution of the loop will consist only of a single instruction. 32-bit and 64-bit integer add, and floating point multiply-accumulate were tested. The results are shown in Figure 38. The theoretical upper bound on integer throughput in the test system is 3 integer ALUs $*$ 4 cores $*$ $2.66 * 10^9$ $cycles/s$ $= 31.2 * 10^9 ops/s$. 32-bit adds come very close to this limit, achieving 81% of the maximum throughput. 64-bit adds achieve roughly half of the maximum throughput. 32-bit floating point multiply-accumulate operations are much slower, only achieving 4GFLOPs on all 4 cores. This is slower than the peak performance of the test system, and more exploration into the generated x86 machine code is needed to understand exactly why. These results suggest that code translated by Ocelot will be relatively fast when performing integer operations, and slow when performing floating point

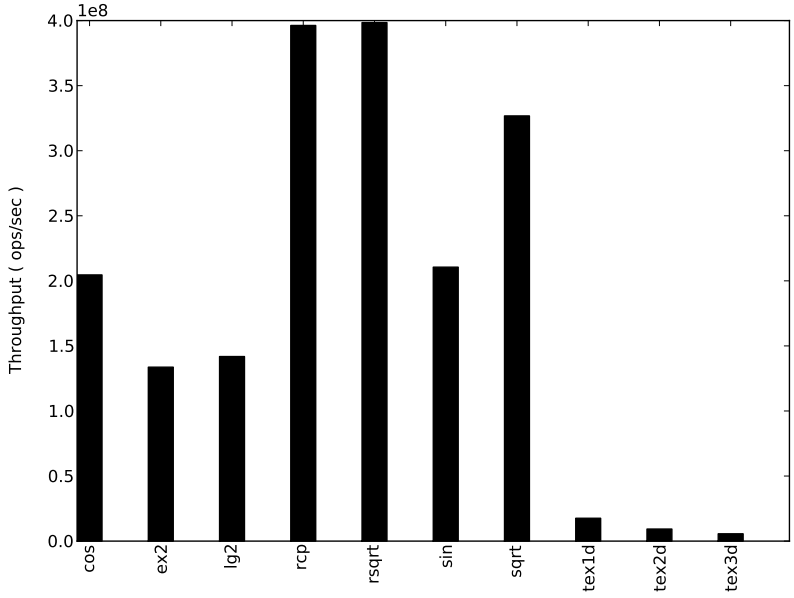

Figure 39: Special instruction throughput. Texture operations are significantly slower without hardware support.

operations, compared to a native implementation.

Special Function Throughput. The final microbenchmark explores the throughput of different special functions and texture sampling. This microbenchmark is designed to expose the maximum sustainable throughput for different special functions, rather than to measure the performance of special functions in any real application. To this end, the benchmarks launch enough CTAs such that there is at least one CTA mapped to each worker thread. Threads are serialized in these benchmarks because there are no barriers, so the number of threads launched does not significantly impact the results. The benchmarks consist of a single unrolled loop per thread where the body consists simply of a series of independent instructions. To determine the benchmark parameters that gave the optimal throughput, the number of iterations and degree of unrolling was increased until less than a 5% change in measured throughput was observed. Eventually, the configuration yielding the greatest throughput was determined to be 16 CTAs, 128 threads per CTA, and 2000 iterations each of which contains a body of 100 independent instructions. Inputs to each instruction were

generated randomly using the Boost 1.40 implementation of Mersenne Twister, with a different seed for each run of the benchmark. The special functions tested were reciprocal (rcp), square-root (sqrt), sin, cos, logarithm base 2 (lg2), 2^{power} (ex2), and 1D, 2D, and 3D texture sampling.

Figure 39 shows the maximum sustainable throughput for each special function. The throughputs of these operations are comparable when run on the GPU, which uses hardware acceleration to quickly provide approximate results. Ocelot implements these operations with standard library functions, incurring the overhead of a fairly complex function call per instruction in all cases except for rcp, which is implemented using a divide instruction. Rcp can be used as a baseline, as it shows the throughput of the hardware floating point divider. Based on these results, the special operation throughput using Ocelot is significantly slower than the GPU, even more so than the ratio of theoretical FLOPs on one architecture to the other. Additionally, the measurements include a significant amount of variance due to the random input values. This is a different behavior than the GPU equivalents, which incur a constant latency per operation.

8.3.2 Runtime Overheads

The next set of experiments were designed to measure the startup cost of each kernel, the overhead introduced by optimizing LLVM code before executing it, and finally the contribution of various translation operations to the total execution time of a program.

Kernel Startup and Teardown. The use of a multi-threaded runtime for executing translated programs on multi-core CPUs introduces some overhead for distributing the set of CTAs onto the CPU worker threads. Ocelot was instrumented using high precision linux timers to try to measure this overhead. Table 4 shows the measured startup and teardown cost for each kernel. Note that the precision of

Application	Startup Latency (s)	Teardown Latency (s)
CP	4.45843e-05	6.07967e-05
MRI-Q	3.48091e-05	8.55923e-05
MRI-FHD	3.62396e-05	8.4877e-05
SAD	4.14848e-05	5.45979e-05
TPACF	3.48091e-05	8.70228e-05
PNS	4.48227e-05	8.53539e-05
RPES	4.17233e-05	6.12736e-05

Table 4: Kernel Startup And Teardown Overhead

these timers is on the order 10us, thus the results indicate that the startup and teardown costs are less than the precision of the timers. These are completely negligible compared to the overheads of translation and optimization. They suggest that there is room for dynamic work distribution mechanisms such as work stealing that have greater overheads.

Optimization Overhead. The next experiment attempts to measure the relative overhead of applying different levels of optimization at runtime by instrumenting the LLVM scalar optimization passes in Ocelot to determine the amount of time spent in optimization routines. The experiment was run on every application in the parboil benchmark suite to identify any differences in optimization time due to the input program's structure. Figure 40 shows that O3 is never more than 2x slower than O1. Optimization for size is almost identical to O2 in all cases, and O3 is only significantly slower than O2 for PNS and SAD.

The best optimization level depends significantly on the application. For CP, MRI-Q, and SAD, the overhead of performing optimizations can not be recovered by improved execution time, and total execution time is increased for any level of optimization. The other applications benefit from O1, and none of the other optimization levels do better than O1 (except for O2 on PNS). Note that the LLVM to x86 JIT always applies basic register allocation, peephole instruction combining, and code scheduling to every program regardless of optimizations at the LLVM level. These

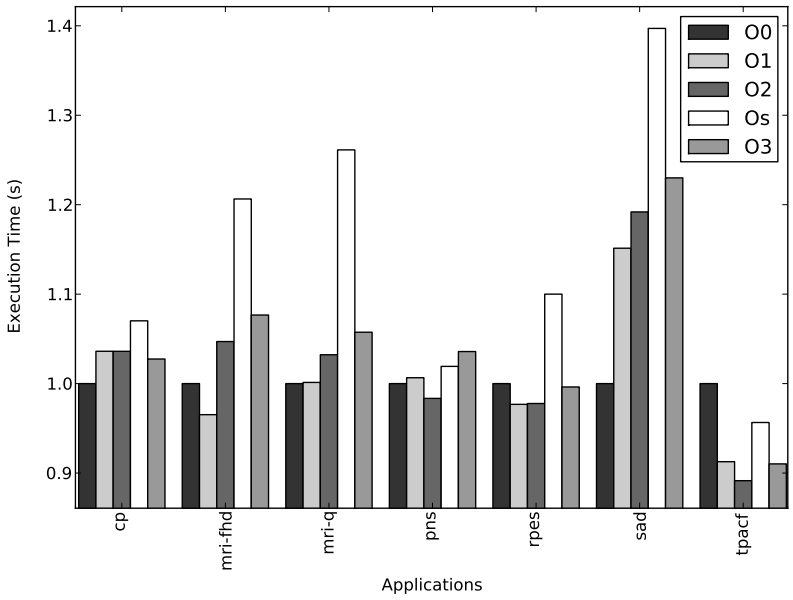

Figure 40: Performance of the same applications with different levels of optimization. The overheads of performing optimizations at runtime must be weighed against the speedup provided by executed more optimized code. The best optimization level is application dependent.

may make many optimizations at the LLVM level redundant, not worth dedicating resources to at execution time. Also, note that the optimizations used here were taken directly from the static optimization tool OPT, which may not include optimizations that are applicable to dynamically translated programs.

Component Contribution. As a final experiment into the overheads of dynamic translation, callgrind [107] was used to determine the relative proportion of time spent in each translation process. Note that callgrind records basic block counts in each process, which may be different than total execution time. Figure 41 shows that the vast majority of the translation time is spent in the LLVM code generator. The decision to use a new LLVM IR in Ocelot only accounts for 6% of the total translation overhead. The time it takes to translate from PTX to LLVM is less than the time needed to parse either PTX or LLVM, and the speed of Ocelot's PTX parser is on par with the speed of LLVM's parser. LLVM optimizations can be a major part of the translation time, but performing PTX-to-PTX transformations in Ocelot takes

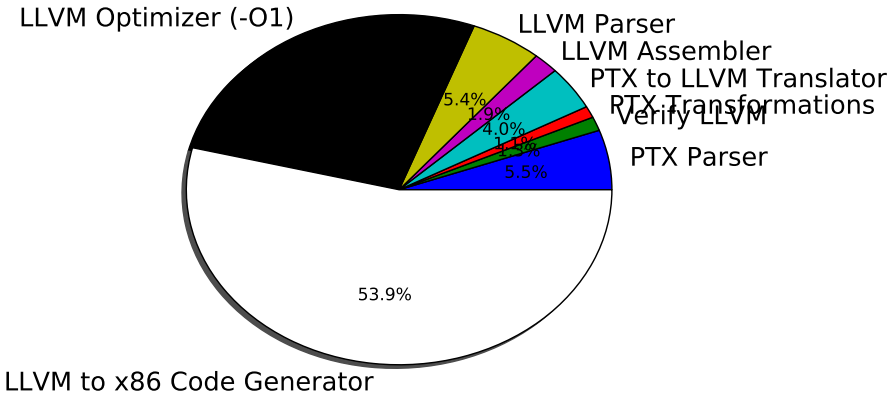

Figure 41: The breakdown of time spent in different Ocelot modules. x86 code generation is the most significant source of overheads.

less than 2% of the total translation time. These results justify many of the design decisions made when implementing Ocelot's CPU back-end.

8.3.3 Full Application Scaling

Moving on from micro-benchmarks to full applications, this experiment studies the ability of CUDA applications to scale to many cores on a multi-core CPU. Test system includes a processor with four cores, each of which supported hyper-threading. Therefore, perfect scaling would allow performance to increase with up to 8 CPU worker threads. This is typically not the case due to shared resources such as caches and memory controllers, which can limit memory bound applications.

The Parboil benchmarks were used as examples of real CUDA applications with a large number of CTAs and threads; previous work on Ocelot shows that the Parboil applications launch between 5 thousand and 4 billion threads per application [84]. Figure 42 shows the normalized execution time of each application using from 1 to 8 CPU worker threads. All of the applications scale well to two threads, but not necessarily beyond that. The CP benchmark is able to achieve better than a 4x speedup using 8 threads, indicating that it is probably compute bound and is able to benefit from hyper-threading. Conversely, SAD slows down when the number of

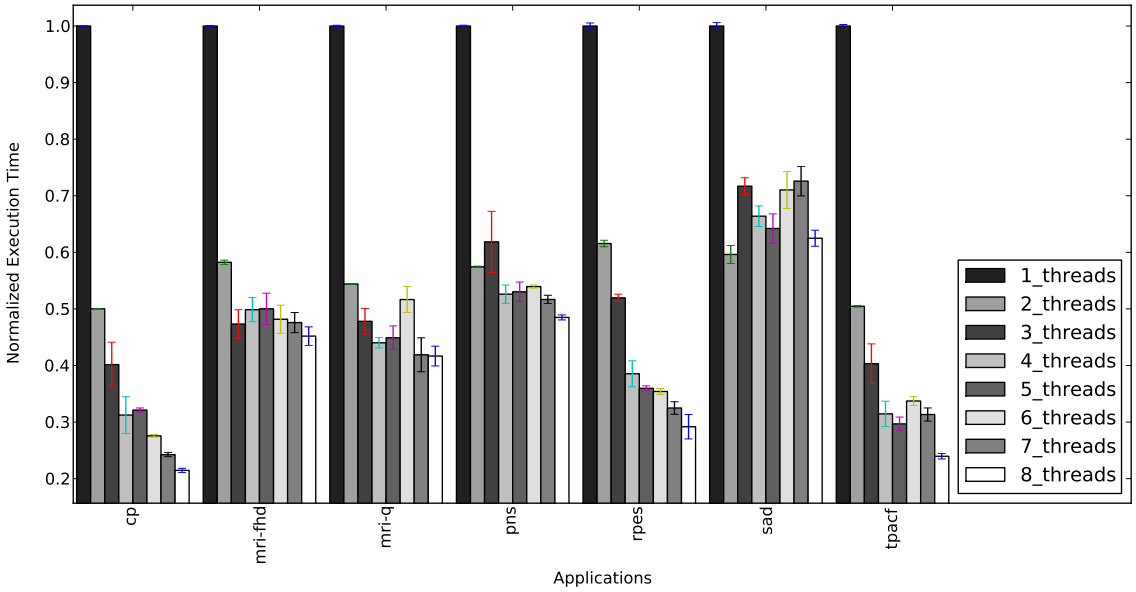

Figure 42: The scaling of applications from the Parboil benchmark suite on 4 cores and 2-way simultaneous multi-threading.

threads is increased beyond two, indicating that the additional threads are probably competing for memory bandwidth and cache occupancy. These results suggest that some applications are significantly more suited to execution on multi-core CPUs than others. In future work, it would be interesting to determine if these trends hold on GPUs as well, or if there are some applications that scale well on GPUs, but poorly on CPUs and visa versa.

8.4 Subkernel Formation

8.4.1 Abstract

Bulk-synchronous programming models are effective mediums for expressing applications with thousands or millions of threads. However, there are several problems associated with conventional compiler and runtime system support for applications with such a large number of threads. First, dynamic compilation and optimization is applied to whole kernels, regardless of the code coverage or size of the application. Second, existing thread scheduling and load balancing techniques are difficult to apply to fine-grained threads without incurring significant runtime overheads. This case study introduces a new program structure, known as the *subkernel*, to address these problems. The subkernel provides structured boundaries for performing dynamic compiler optimizations and runtime scheduling of bulk-synchronous threads. Preliminary experiments show that the subkernel is highly efficient for a range of existing multi-core and SIMD processors.

8.4.2 Introduction

There are two major problems associated with applying traditional dynamic compilation and thread scheduling techniques to bulk-synchronous applications with millions of threads. First, lazy dynamic compilers that switch between compilation and execution of an application translate basic blocks at a time. For multi-threaded programs, this requires a global lock around the dynamic compiler (if it is not reentrant), as well as either i) a coarse grained lock on the compiled code cache, or ii) fine-grained locks on the blocks being translated. This is very effective in the case of a single or modest number of threads. However, bulk-synchronous applications typically execute thousands or millions of lightweight threads. Lock contention and serialization through the compiler significantly limit the performance of these approaches. For this

reason, no existing dynamic compilers for bulk-synchronous applications use this approach. Instead, they typically compile the entire reachable call-graph from the entry point of a bulk-synchronous kernel, introducing a significant compilation overhead for applications with large code size or poor code coverage.

The second problem is that existing runtime thread scheduling techniques do not fit well with existing thread serialization techniques or the locality requirements of multi-core SIMD processors. Thread serialization refers to the process of fusing the execution of multiple threads together into a single executable unit using a compiler transformation. In the context of bulk-synchronous applications, thread serialization refers to the process of combining many light-weight threads together into a single heavy-weight thread that performs the same function. Thread serialization is an important method for controlling the overheads of managing thousands or millions of threads on processors with tens or hundreds of cores.

This section proposes the use of a structure, known as the *subkernel*, to address these two problems. A subkernel is a set of basic blocks in a program control flow graph with multiple entry points, multiple exit points, and embedded thread scheduling logic. Subkernels are formed using a combination of trace formation and thread serialization. Basic blocks are included based on their profiled execution characteristics, control flow structure, and use of inter-thread synchronization operations. Lazy dynamic compilation is enabled by grouping threads together at subkernel entry points, and batch scheduling them in an interleaved manner with code generation or optimization passes, which operate on subkernels at a time. Thread serialization is performed within subkernels, and runtime thread scheduling can be performed at subkernel boundaries. The granularity of subkernels can be coarsened or refined to balance the overheads of dynamic compilation and fine-grained scheduling against the benefits of hot-path optimizations and balanced, locality-aware, thread schedules.

8.4.3 Bulk-Synchronous Models

8.4.4 The Subkernel

A subkernel is a set of strongly connected basic blocks for which control may both enter and exit through multiple points, and which include embedded logic for thread scheduling within the subkernel. The entry points to a subkernel are merged together into a single scheduler block, from which a batched group of threads enters the subkernel as a single unit. Once in a sub-kernel, threads may execute in near lock step, or they may take divergent paths through the subkernel, eventually leaving the subkernel through one of the exit points. The purpose of a subkernel is twofold. First, it defines explicit regions that can be used to amortize the costs of dynamic compilation and thread scheduling. Second, it creates blocked code regions in which a group of threads can be encouraged to execute in near lock-step, increasing instruction spatial and temporal locality and introducing opportunities for local thread coalescing and serializing optimizations. All basic blocks between synchronization or yield points are not included in subkernels. Instead, blocks are heuristically included in the subkernel to allow for more efficient thread scheduling within the subkernel, and to balance compiler and thread scheduling overheads.

A thread serialization loop is the set of blocks along all shortest control paths between synchronization points in the program. Thread serialization loops are a special of subkernels that are formed between synchronization points in the program. Subkernels, on the other hand, are formed selectively from a set of basic blocks that are expected to balance between thread scheduling and dynamic compilation overheads. Thus, they provide a more flexible structure for implementing dynamic compiler and runtime thread scheduler optimizations.

This section discusses block selection for subkernels, subkernel formation, generation of thread scheduling logic within a subkernel, and extensions of existing techniques for dynamic compilation and thread scheduling using subkernels.

Subkernel Block Selection Subkernels are formed from the basic blocks in the program control flow graph. It is also useful, but not necessary, to have access to the program call graph to optimize call sites that occur within subkernels. The first step in subkernel formation involves selecting a set of blocks that should be included in the subkernel. This region of blocks typically corresponds to a structured subgraph of the program control tree without function calls, indirect branches, or exceptions. However, other regions, including the headers and bodies of arbitrarily nested loops, conditional statements, or partially in-lined function calls may also be chosen. Existing techniques for thread serialization [61, 137] form regions that originate and end at explicit thread synchronization points. In contrast, subkernels are formed from the subset of blocks that make good candidates for thread scheduling and dynamic compilation.

Subkernel Formation The heuristics used to form subkernels are influenced by whether subkernel formation is performed by a static or a dynamic compiler. One of the primary goals of subkernel formation is to limit the overheads of dynamic compilation by performing code generation and optimization lazily as subkernels are executed. When used by a dynamic compiler, the block selection and subkernel formation algorithms should run significantly faster than the code generation and optimization passes to provide a performance benefit. A simple and fast algorithm for block selection involves performing a bread-first traversal of the program, tracking all of the paths from the entry point of the current subkernel to all entry points. When the minimum path length crosses a pre-defined threshold, a sub-kernel is created and the process is repeated after selecting a new candidate for the entry point to the next subkernel. This algorithm only requires visiting each basic block once.

Subkernel Thread Scheduling Thread scheduling overheads can be removed by embedding the thread scheduling mechanism in the subkernel. Embedding the

scheduling mechanism removes overhead in two ways: i) it allows the use of a simpler scheduling algorithm within a subkernel, and ii) it allows a specialized scheduler to be generated that only handles the operations that can be performed in a specific subkernel. For example, a thread context does not need a reference to the stack pointer if stack is not accessed within the subkernel. Three thread scheduling optimizations are considered for improving the efficiency of subkernels: i) embedded thread loops, ii) thread vectorization, iii) and static thread schedules. They are described in the following section.

The first simplified scheduling algorithm iterates over a batch of threads that enter the subkernel by inserted the loop directly into the subkernel. A useful optimization involves requiring the higher level scheduler to ensure that threads are issued in groups with affine thread ids. In this case, a scheduling loop can be placed over the subkernel that iterates over a register mapped to the thread id. If the batch-size is fixed statically, then the trip count of the loop can be set as a constant value, and the subkernel can be aggressively optimized.

Another approach involves and explicitly issuing vector instructions that perform the same operations as a batch of scalar threads. This is equivalent to scheduling the threads to execute in a lock-step, SIMD manner. Potentially divergent branches can include checks for divergence and conditional branches to exit points from the subkernel if the branch is found to be divergent at execution time. This allows for SIMD execution of potentially divergent control flow on architectures that do not include hardware support for handling branch divergence within a SIMD group of threads.

The third scheduling technique involves explicitly interleaving the instruction streams from multiple threads. This technique can be used to harvest independent instructions from different threads to fill functional unit slots in wide superscalar or VLIW architectures. If the target architecture supports predicated execution, then

multiple control paths from different threads can be merged together to handle divergence. Additionally, divergent control flow across multiple threads can be handled by enumerating all possible combinations of control paths between divergent branches and re-convergence points and using an indirect branch where the target is determined from the combined branch conditions of all of the threads. This technique can be used to remove all overheads of branch divergence. However, the combinatorial explosion of possible paths for each thread limits its application to a moderate number of threads.

8.4.5 Performance Evaluation

This section evaluates the efficacy of subkernels within a dynamic compilation framework for a set of benchmarks from diverse application domains.

Methodology The subkernel concepts presented in this case study have been implemented and are being currently deployed in the Ocelot compiler. Ocelot is a mid-level dynamic compiler for the NVIDIA PTX virtual ISA that is designed to generate efficient code for multi-core SIMD processors such as x86+Altivex CPUs, NVIDIA Fermi GPUs, and AMD Sourthern Island GPUs. Optimizations are performed on the mid-level virtual ISA, and then lowered down to the native ISA of the various backends. Processor targets typically require close interaction between the compiler and a lightweight runtime layer that is responsible for thread scheduling and state management. For the CPU targets, this support is implemented using a lightweight software layer. For the GPU targets, this runtime functionality is split between a software layer, and a series of hardware state machines that are programmed using a driver interface.

Subkernel Size Figure 43 shows the change in execution time of varying the subkernel size on an individual CUDA benchmark. The leftmost section of the figure

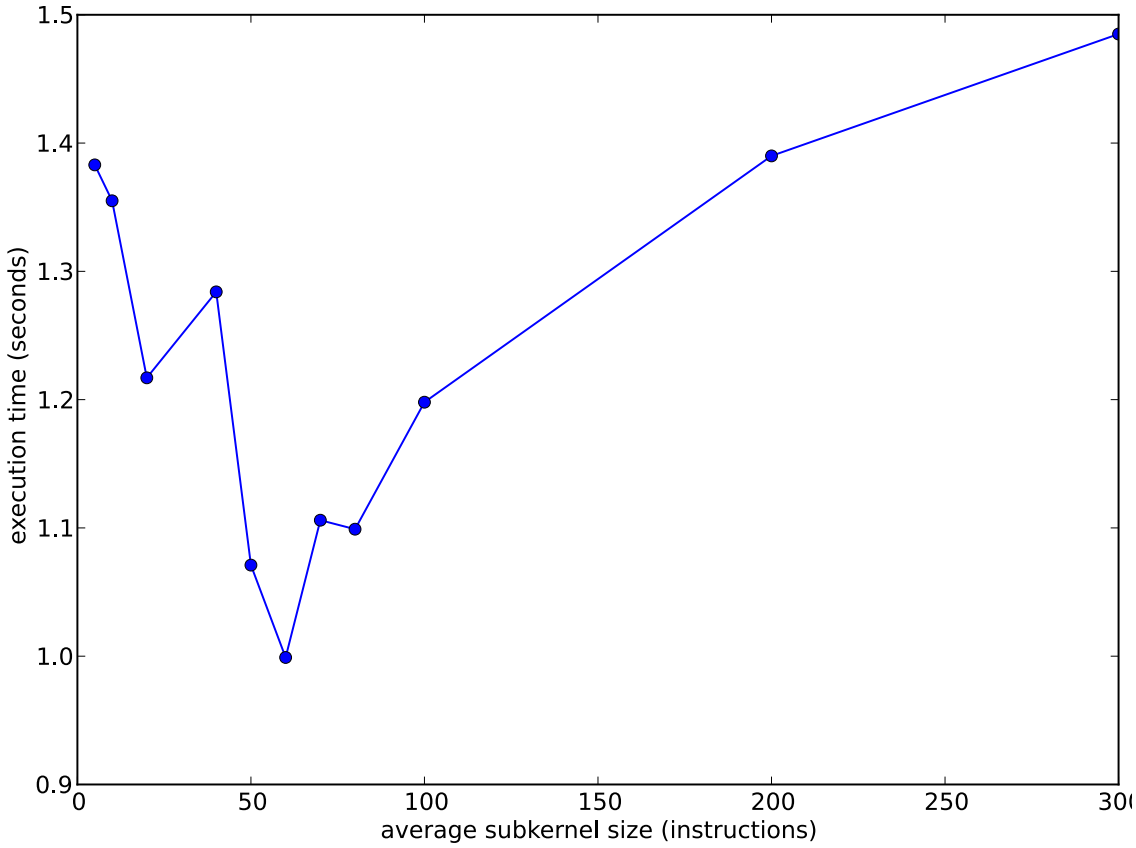

Figure 43: The change in execution time of a CUDA benchmark with varying subkernel sizes.

shows an extreme case where every instruction is its own subkernel. In this case, only those instructions that are actually executed will be compiled. The rightmost section of the figure shows the other extreme where the entire benchmark is compiled at once. Performance is maximized in the middle of the figure for subkernel sizes of about 65 instructions, where the overheads of dynamic compilation are balanced with the overheads of frequent scheduling.

Thread Serialization Different methods for thread serialization within subkernels are evaluated. Vectorization involves packing groups of eight threads together into AVX instructions, interleaving involves interleaving the instruction streams from multiple threads, and embedded thread loops involve explicitly looping over each thread.

8.4.6 Concluding Remarks

Existing solutions for dynamic compilation and runtime scheduling for bulk-synchronous programming models with fine-grained threads have two major problems: i) dynamic compilation and optimization is performed on entire kernels, and ii) thread serialization techniques are not compatible with existing methods for thread scheudling. This case study introduces the subkernel program structure to address these issues. Subkernels are created by selecting strongly connected basic blocks based on their profiled execution characteristics, instruction characteristics, and use of inter-thread synchronization operations. Lazy dynamic compilation is enabled by grouping threads together at subkernel entry points, and batch scheduling them in an interleaved manner with code generation or optimization passes, which operate on subkernels at a time. Thread serialization is performed within subkernels, and runtime thread scheduling can be performed at subkernel boundaries. Preliminary experiments show that the subkernel is highly efficient for a range of existing bulk-synchronous applications executing on multi-core SIMD processors.

8.5 On-line Kernel Performance Modeling

This case study reports on an empirical evaluation of 25 CUDA applications on four GPUs and three CPUs, leveraging the Ocelot dynamic compiler infrastructure which can execute and instrument the same CUDA applications on either target. Using a combination of instrumentation and statistical analysis, we record 37 different metrics for each application and use them to derive relationships between program behavior and performance on heterogeneous processors. These relationships are then fed into a modeling framework that attempts to predict the performance of similar classes of applications on different processors. Most significantly, this study identifies several non-intuitive relationships between program characteristics and demonstrates that it is possible to accurately model many aspects of expected kernel performance using only metrics that are available before a kernel is executed.

As programming models such as NVIDIA's CUDA [110] and the industry-wide standard OpenCL [57] gain wider acceptance, efficiently executing the expression of a data-parallel application on parallel architectures with dissimilar performance characteristics such as multi-core super-scalar processors or massively parallel graphics processing units (GPUs) becomes increasingly critical. While considerable efforts have been spent optimizing and benchmarking applications intended for processors with several cores, comparatively less effort has been spent evaluating the characteristics and performance of applications capable of executing efficiently on both GPUs and CPUs.

As more applications are written from the ground up to perform efficiently on systems with a diverse set of available processors, choosing the architecture capable of executing each kernel of the application most efficiently becomes more important in maximizing overall throughput and power efficiency. At the time of this writing, we are not aware of any study that correlates detailed application characteristics with actual performance measured on real-world architectures.

Consequently, this section makes the following contributions:

- We identify several PTX application characteristics that indicate relative performance on GPUs and CPUs.
- We use a statistical data analysis methodology based on principal component analysis to identify critical program characteristics.
- We introduce a model for predicting relative performance when the application is run on a CPU or a GPU.

The analysis and models presented in this paper leverage the Ocelot framework for instrumenting data-parallel applications and executing them on heterogeneous platforms. The Ocelot framework [84] is an emulation and compilation infrastructure that implements the CUDA Runtime API and either (1) emulates executing kernels, (2) translates kernels to the CPU ISA, or (3) emits the kernel representation to the CUDA driver for execution on attached GPUs. Ocelot is uniquely leveraged to gather instruction and memory traces from emulated kernels in unmodified CUDA applications, analyze control and data dependences, and execute the kernel efficiently on both CUDA-capable GPUs and multicore CPUs.

8.5.1 Instrumentation With Ocelot

The Ocelot compiler infrastructure strives to decouple CUDA applications from GPUs by wrapping CUDA Runtime API [110], parsing kernels stored as assembly text within the application into an internal representation, executing these kernels on devices present, and maintaining a complete list of CUDA memory allocations that store state on the GPU. By decoupling a CUDA application from the CUDA driver, Ocelot provides a framework for simulating GPUs, gathering performance metrics, translating kernels to architectures other than GPUs, instrumenting kernels, and optimizing kernels for execution on the GPU.

Ocelot's translation framework provides efficient execution of CUDA kernels on multi-core CPUs by first translating the kernel from PTX to the native instruction set of the Low-Level Virtual Machine (LLVM) [92] infrastructure then applying a series of transformations that implement the PTX execution model discussed in Section 7.1.4 with control and data structures available to typical scalar microprocessors. The complete translation process is beyond the scope of this section; we only include a brief description of the process as it pertains to the analysis methodology used in the following sections. A more detailed overview of the translation process was covered previously in chapter 8.3.0.1.

8.5.2 Metrics and Statistics

Ocelot's PTX emulator may be instrumented with a set of user-supplied event handlers to generate detailed traces of instructions and memory references. After each dynamic PTX instruction is completed for a given program counter and set of active threads, an event object containing program counter, PTX instruction, activity mask, and referenced memory addresses is dispatched to each registered trace generator which handles the event according to the performance metric it implements. The following application metrics presented were gathered in this manner building on the set of metrics defined in previous work [84]:

Activity Factor Any given instruction is executed by all threads in a warp. However, individual threads can be predicated off via explicit predicate registers or as a result of branch divergence. Activity factor is the fraction of threads active averaged over all dynamic instructions.

Branch Divergence When a warp reaches a branch instruction, all threads may branch or fall through, or the warp may diverge in which case the warp is split with some threads falling through and other threads branching. Branch Divergence is

the fraction of branches that result in divergence averaged over all dynamic branch instructions.

Instruction Counts These metrics count the number of dynamic instructions binned according to the functional unit that would execute them on a hypothetical GPU. The functional units considered here include integer arithmetic, floating-point arithmetic, logical operations, control-flow, off-chip loads and stores, parallelism and synchronizations, special and transcendental, and data type conversions.

Inter-thread Data Flow The PTX execution model includes synchronization instructions and shared data storage accessible by threads of the same CTA. Inter-thread data flow measures the fraction of loads from shared memory such that the data loaded was computed by another thread within the CTA. This is a measure of producer-consumer relationships among threads.

Memory Intensity Memory intensity computes the fraction of instructions resulting in communication to off-chip memory. These may be explicit loads or stores to global or local memory, or they may be texture sampling instructions. This metric does not model the texture caches which are present in most GPUs and counts texture samples as loads to global memory.

Memory Efficiency Loads and stores to global memory may reference arbitrary locations. However, if threads of the same warp access locations in the same block of memory, the operation may be completed in a single memory transaction; otherwise, multiple transactions are created and serialized. This metric expresses the minimum number of transactions needed to satisfy every dynamic load or store divided by the actual number of transactions, computed according to the memory coalescing protocol defined in [110] §5.1.2.1. This is a measure of spatial locality.

Memory Extent This metric uses pointer analysis to compute the working set of kernels as the number and layout of all reachable pages in all memory spaces. It represents the total amount of memory that is accessible to a kernel immediately before it is executed.

Context Switch Points CTAs may synchronize threads at the start and end of kernels as well as within sections of code with uniform control flow, typically to ensure shared memory is consistent when sharing data. Each synchronization requires a context switch point inserted by Ocelot during translation for execution on multicore as described in [41].

Live Registers Unlike CPUs, GPUs are equipped with large register files that may store tens of live values per thread. Consequently, executing CTAs on a multicore x86 CPU requires spilling values at context switches. This metric expresses the average number of spilled values.

Machine Parameters GPUs and CPUs considered here are characterized by clock rate, number of concurrent threads, number of cores, off-chip bandwidth, number of memory controllers, instruction issue width, L2 cache capacity, whether they are capable of executing out-of-order, and the maximum number of threads within a warp.

Registers per Thread The large register files of GPUs may be partitioned into threads at runtime according to the number of threads per CTA. Larger numbers of threads increases the ability to hide latencies but reduces the number of registers available per thread. On CPUs, these may be spilled to local memory, which is mapped into the data cache. This metric expresses the average number of registers allocated per thread.

Application	Full Name	Source
MRI-Q	Magnetic Resonance Imaging	Parboil
MRI-FHD	Magnetic Resonance Imaging	
CP	Coulombic Potential	
SAD	Sum of Absolute Differences	
TPACF	Two-Point Angular Correction	
PNS	Petri Net Simulation	
RPES	Rys Polynomial Equation Solver	
hotspot	Thermal simulation	Rodinia
lu	Dense LU Decomposition	
nbody	Particle simulation	CUDA SDK

Table 5: Benchmark Applications.

Kernel Count The number of times an application launches a kernel indicates the number of global barriers across all CTAs.

Parallelism Scalability This metric determines the maximum amount of SIMD and MIMD parallelism [84] available in a particular application averaged across all kernels.

DMA Transfer Size CUDA applications explicitly copy buffers of data to and from GPU memory before kernels may be called incurring a latency and bandwidth constrained transfer via the PCI Express bus of the given platform. We measure both the number of DMAs and the total amount of data transferred.

8.5.3 Benchmarks

For this study, we selected applications from existing benchmark suites. PARBOIL [72] consists of seven application-level benchmarks written in CUDA that perform a variety of computations including ray tracing, finite-difference time-domain simulation, sorting. Rodinia [27] is a separate collection of applications for benchmarking GPU systems. Finally, the CUDA SDK is distributed with over fifty applications showcasing CUDA features. A list of the applications we selected appears in Table 5.

Kernels from these applications were executed on processors whose parameters are summarized in Table 6. This selection consists of both CPUs and GPUs that together offer a wide range for each of the listed parameters. We expect these parameters that capture clock frequency, issue width, concurrency, memory bandwidth, and cache structure to sufficiently model the performance of kernels from the benchmark applications.

We chose to characterize PTX applications by the collection of statistics listed in Table 7. These may be classified according to the way they are gathered. Some quantities may be determined via *static analysis* before a kernel is executed such as static instruction counts of each kernel, the number/size of DMA operations initiated before a kernel launch, as well as upper bounds on working set size determined by conservative pointer analysis. Others may be determined at runtime by inserting *instrumentation* into kernels and recording averages as they execute; these include SIMD and MIMD parallelism metrics. Finally, some metrics – typically dynamic instruction counts – may only be determined by executing the kernel to completion via PTX *emulation* and analyzing the resulting instruction traces. Note that all of these metrics were collected via execution on the Ocelot PTX Emulator, therefore, they are independent of the micro-architecture of a particular CPU or GPU.

Table 8 lists the quantitative results from each Parboil application collected for the statistics listed in Table 7. Our analysis also covered the Rodinia benchmarks and the CUDA SDK.

8.5.4 Results

Our methodology for modeling the interaction between machine and program characteristics uses principal component analysis to identify independent parameters, similar to [79], cluster analysis to discovers sets of related applications, and multivariate regression combined with projections onto convex sets [156] to build predictive models

	Nehalem	Atom	Phenom	8600 GS	8800 GTX	GTX280	C1060
Type	Out-of-order CPU	In-order CPU	Out-of-order CPU	In-order GPU	In-order GPU	In-order GPU	In-order GPU
Issue Width	4	2	3	1	1	1	1
Clock Frequency (GHz)	2.6	1.6	2.2	1.2	1.5	1.3	1.3
Hardware Threads per Core	2	2	1	24	24	24	24
Cores	4	1	4	2	16	30	30
Warp Size	1	1	1	32	32	32	32
Memory Controllers	3	1	2	2	6	8	8
Bandwidth per Controller (GB/s)	8.53	3.54	8.53	5.6	14.3	17.62	12.75
L2 Cache (kB)	512	512	512	0	0	0	0

Table 6: Machine parameters.

Metric	Units	Description	Collection method
Extent_of_Memory	bytes	Size of working set	static analysis
Context_switches	switch points	Number of thread context switch points	static analysis
Live_Registers	registers	Number registers spilled at context switch points	static analysis
Registers_Per_Thread	registers	Number of registers per thread	static analysis
DMAs	transfers	Number of transfers between GPU memory	static analysis
Static_Integer_arithmetic	instructions	Number of integer arithmetic instructions	static analysis
Static_Integer_logical	instructions	Number of logical instructions	static analysis
Static_Integer_comparison	instructions	Number of integer compare instructions	static analysis
Static_Memory_offchip	instructions	Number of off-chip memory transfer instructions	static analysis
Static_Memory_onchip	instructions	Number of on-chip memory transfer instructions	static analysis
Static_Control	instructions	Number of control-flow instructions	static analysis
Static_Parallelism	instructions	Number of parallelism instructions	static analysis
Dynamic_Integer_arithmetic	instructions	Number of executed integer arithmetic instructions	emulation
Dynamic_Integer_logical	instructions	Number of executed integer logical instructions	emulation
Dynamic_Memory_offchip	instructions	Number executed off-chip memory transfer instructions	emulation
Dynamic_Memory_onchip	instructions	Number of executed on-chip memory transer instructions	emulation
Dynamic_Integer_comparison	instructions	Number of executed integer comparison instructions	emulation
Static_Float_single	instructions	Single-precision floating point arithmetic	static analysis
Static_Float_comparison	instructions	Single-precision floating point compare	static analysis
Static_Special	instructions	Special function instructions	static analysis
Memory_Efficiency	percentage	Memory efficiency metric	instrumentation
Memory_Sharing	percentage	Inter-thread data flow metric	instrumentation
Activity_Factor	percentage	Activity factor metric	instrumentation
MIMD	speedup	MIMD Parallelism metric	instrumentation
SIMD	speedup	SIMD Parallelism metric	instrumentation
Dynamic_Float_single	instructions	Number of single-precision arithmetic instructions	emulation
Dynamic_Float_comparison	instructions	Number of singl-precision comparison instructions	emulation
DMA_Size	bytes	Avg DMA transfer size	static analysis
Dynamic_Control	instructions	Number of executed control-flow instructions	emulation
Dynamic_Parallelism	instructions	Number of executed parallelism instructions	emulation
Dynamic_Special	instructions	Number of executed spcial instructions	emulation
Static_Float_double	instructions	Number of double precision floating point instructions	static analysis
Memory_Intensity	instructions	Memory Intensity metric	instrumentation
Dynamic_Float_double	instructions	Number of executed double-precision floating point instructions	emulation
Dynamic_Other	instructions	Other instructions	emulation

Table 7: List of metrics.

from principal components.

8.5.4.1 Principal Component Analysis

Principal Component Analysis (PCA) is predicated on the assumption that several variables used in an analysis are correlated, and therefore measure the same property of an application or processor. PCA derives a set of new variables, called principal components, from linear combinations of the original variables such that there is no correlation between any of the new variables. PCA identifies the new variables with

Metric	CP	MRI_FHD	MRI-Q	PNS	RPES	SAD	TPACF
Static							
Extent_of_Memory	1112592	582630	517229	720002676	59883768	8948776	5009948
Context_switches	0	0	0	19	5	2	4
Live_Registers	0.00000	0.00000	0.00000	23.15000	4.60000	5.50000	14.50000
Total_Registers	20.000	18.500	17.000	35.000	27.000	21.000	22.000
DMAs	11	17	11	224	6	3	3
DMA_Size	1.5351e+05	4.6501e+04	6.7397e+04	7.1420e+01	1.1537e+07	2.9996e+06	1.6602e+06
Integer_arithmetic	3410	31080	5388	13102152	200196	1620	448
Integer_logical	0	12516	2136	2300592	111150	280	72
Integer_comparison	220	8582	1400	2244480	25560	142	68
Float_single	5280	17290	2908	617232	1279008	186	18
Float_double	0	840	0	0	0	0	0
Float_comparison	0	1050	180	0	38448	0	6
Memory_offchip	1760	2478	468	1094184	10530	66	14
Memory_onchip	770	1862	332	1178352	142434	198	64
Control_instr	660	11466	1968	3226440	186300	182	134
Parallelism_instr	0	0	0	533064	15030	12	8
Special_instr	880	0	0	0	43254	0	0
Othe_instrr	0	0	0	0	0	0	0
Instrumented							
Activity_Factor	100.000	100.000	100.000	97.200	63.850	95.400	80.510
Memory_Intensity	0.010000	0.060000	0.040000	4.640000	2.740000	5.880000	0.010000
Memory_Efficiency	49.200	49.600	49.600	65.600	73.100	47.700	48.100
Memory_Sharing	0.00000	0.00000	0.00000	51.50000	76.60000	2.90000	12.40000
SIMD Parallelism	128.000	292.570	320.000	248.880	40.580	70.280	206.110
MIMD Parallelism	2.5600e+02	1.1057e+02	9.7500e+01	1.7990e+01	6.4757e+04	5.9400e+02	1.5663e+02
Emulated							
Integer_arithmetic	2.2596e+08	3.7218e+07	2.2805e+07	5.2469e+10	2.1425e+10	1.5161e+07	1.3488e+09
Integer_logical	0	1.3738e+07	7.8520e+06	2.4229e+10	8.9300e+09	5.9380e+05	2.2153e+08
Integer_comparison	1.1267e+08	2.6135e+07	1.2572e+07	5.6280e+09	5.0089e+09	6.0093e+05	2.0857e+08
Float_single	2.9293e+09	2.1333e+08	1.1878e+08	2.1047e+10	4.1966e+10	6.0445e+06	6.0892e+07
Float_double	0	11010048	0	0	0	0	0
Float_comparison	0	1.3738e+07	7.8505e+06	0	1.2119e+09	0	1.4098e+08
Memory_offchip	4.5056e+05	3.8136e+04	1.0896e+04	5.3392e+09	6.8786e+08	1.9127e+05	2.5571e+05
Memory_onchip	4.5064e+08	1.3801e+07	6.3024e+06	4.8278e+08	1.4313e+10	4.4984e+06	3.2223e+08
Control_instr	1.1278e+08	4.5258e+07	2.6641e+07	5.7445e+09	1.5469e+10	8.2229e+05	8.4251e+08
Parallelism_instr	0	0	0	4.1073e+07	2.6514e+09	1.9008e+04	2.0161e+07
Special_instr	9.0112e+08	0	0	0	9.9957e+08	0	0
Other_instr	0	0	0	0	0	0	0

Table 8: Metrics for each of the Parboil benchmark applications using default input sizes.

the most information about the original data thereby reducing the total number of variables needed to represent a data set. In our analysis, we use a normalized PCA (zero mean, unit variance) because each of our original metrics are expressed using different units. We choose enough principal components to account for at least 85% of the variance in the original data.

Once PCA has identified a set of principal components, we apply a varimax rotation [81] to the principal components. This distributes the contribution of each original variable to each principal component, such that each original variable either

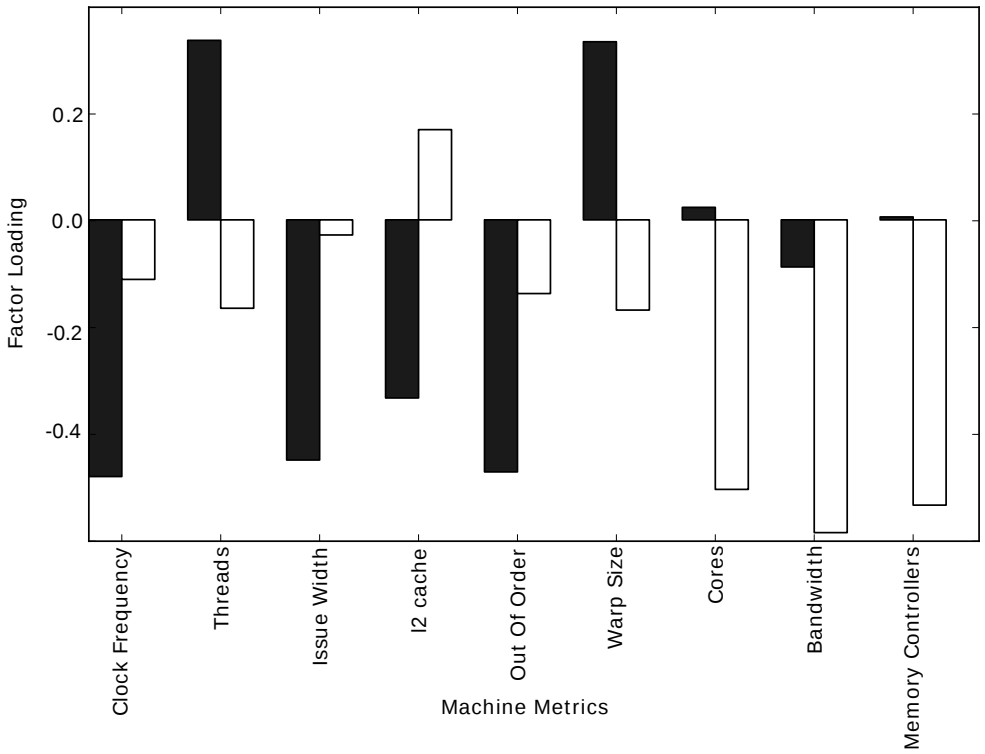

Figure 44: Factor loadings for two machine principal components. PC0 (black) corresponds to single core performance, while PC1 (white) corresponds to multi-core throughput.

strongly impacts a principal component or it very weakly impacts it. In other words, it causes each original metric to influence a single principal component, easing analysis of the data.

For the statistics gathered in the previous section, we perform two separate PCAs: one which only includes application statistics and another which only includes machine statistics. This is valid because the metrics were collected via the Ocelot PTX emulator, which is architecture agnostic.

8.5.4.2 Machine Principal Components

From the set of machine statistics, PCA yielded two principal components that are shown in terms of factor loadings in Figure 44 and plotted in Figure 45. Clusters reiterate the few number of high-speed cores among the CPUs and a much larger

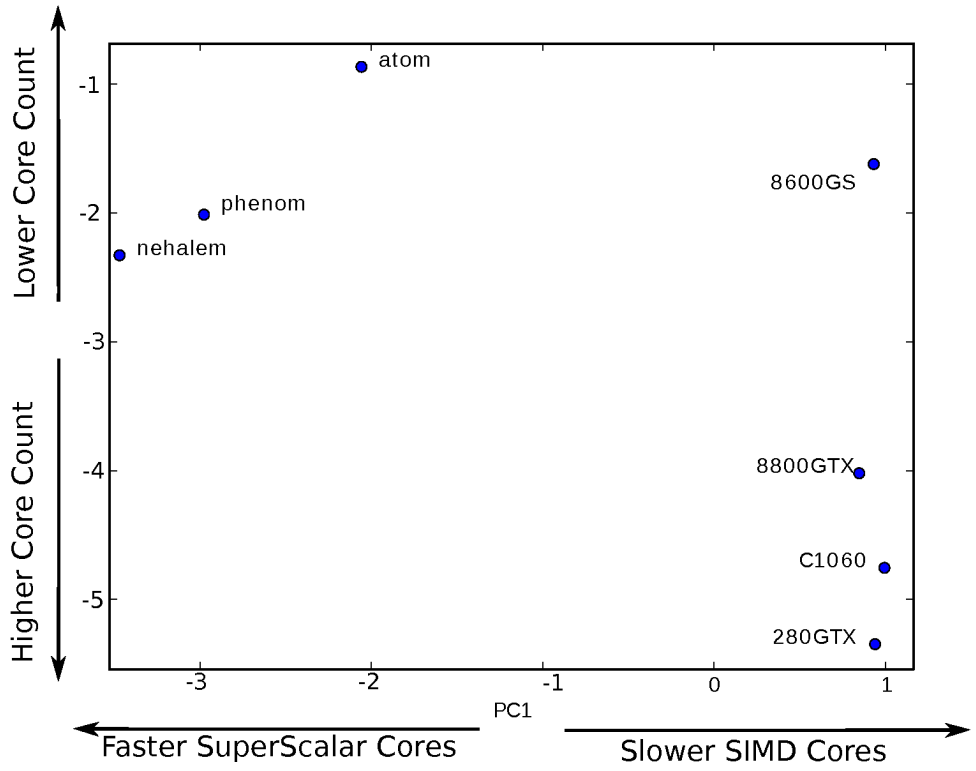

Figure 45: The machine principal components. GPUs have high core counts and slow SIMD cores while CPUs have fewer, but faster, cores.

number of lower-speed cores among the GPUs.

PC0: Single Core Performance The variables that contribute strongly to the first principal component are shown in the left of Figure 44. Note that all of these metrics, clock frequency, issue width, cache size, etc correspond to the performance of a single processor core. Additionally, note that threads-per-core and warp size are negatively correlated with clock frequency, issue width, and out of order, highlighting the differences between GPU and CPU design philosophies.

PC1: Core and Memory Controller Count The second PC illustrates that the core count is correlated with the memory controller count and memory bandwidth per channel, indicating that multi-core CPUs and GPUs are designed such that the off-chip bandwidth scales with the number of cores.

Discussion Though the intent of this paper is to derive relationships between these metrics and the performance of machine-application combinations, this analysis also exposes trends in the way that CPUs and GPUs are designed. The division of the machine metrics into these two principal components can be explained as follows: the design of a single core typically does not influence the synthesis of many single cores and memory controllers into a multi-core processor. The performance of a single core in a processor is either characterized by clock frequency, cache size, superscalar width, etc or a high degree of hardware multithreading and large SIMD units. Figure 45 shows a clear distinction between CPU and GPU architectures.

This classification holds even for processors not included in this study. For example, recently released GPUs by Intel [75] and AMD can both be characterized by a large number of threads per core and wide SIMD units; even embedded CPUs such as ARM Cortex A9 [36] have begun to move towards out of order execution.

8.5.4.3 Application Components

The PCA of the application statistics yielded five principal components, the factor loadings of which are shown in Figure 46. We would like to note that PCA reveals relationships that hold only for a given set of data, in this case, the applications that were chosen. Given a different set of applications, PCA may reveal a different set of relationships. However, the fact that these trends are valid across application from both Parboil and Rodinia, which are designed to be representative of CUDA applications, indicates that they may represent fundamental similarities in the way that developers write CUDA programs.

PC0: MIMD Parallelism . The first principal component is composed of metrics that are related to the MIMD parallelism of a program. Recall that MIMD parallelism measures the speedup of a kernel on an idealized GPU with an infinite number of cores and zero memory latency. It is bound by the number of CTAs in each kernel. The

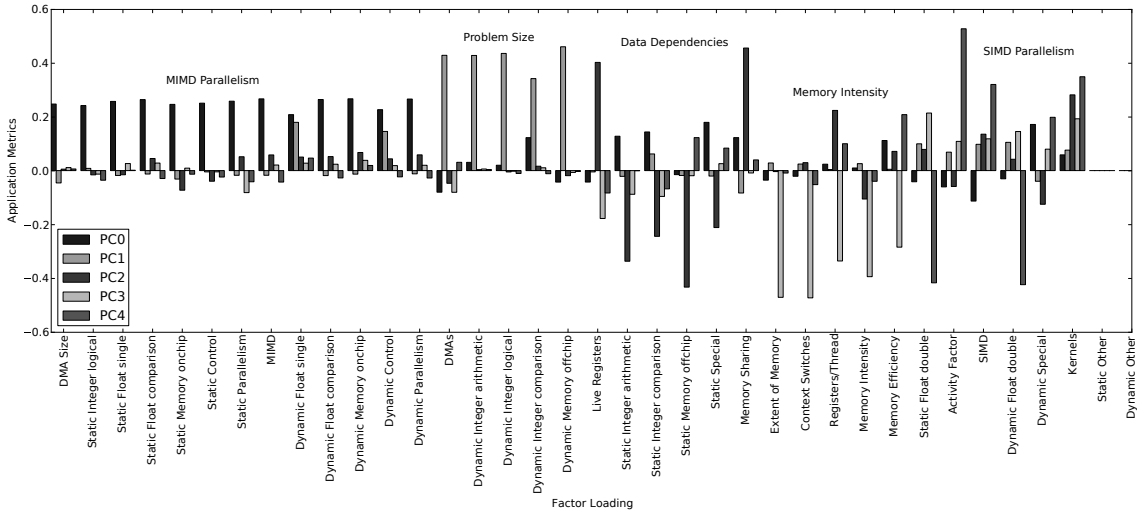

Figure 46: Factor loadings for the five application principal components. A factor loading closer to +-1 indicates a higher influence on the principal component.

correlation between MIMD parallelism and DMA Size indicates that applications that copy a larger amount of memory to the GPU will also launch a large number of CTAs. It is interesting to note that during our preliminary evaluation which only included the Parboil benchmarks shown in Table 8, this component also included the majority of dynamic instruction counts. Adding the Rodinia *hotspot* application and the SDK *nbody* application broke the relationship between MIMD parallelism and problem size, indicating that not all CUDA applications are weakly scalable. This distinction motivates the need for the cluster analysis in the next section, where applications with similar characteristics can be identified and modeled separately. As a final point, notice that several static instruction counts are highly correlated with the problem size. This relationship is difficult to explain intuitively, and it would be relatively simple to craft a synthetic application that breaks this relationship. However, it is significant that none of these applications do. Results like this motivate the use of a technique like PCA, which is able to discover relationships that defy intuition.

PC1: Problem Size . The second component is composed most significantly of average dynamic integer, floating point, and memory instruction counts which

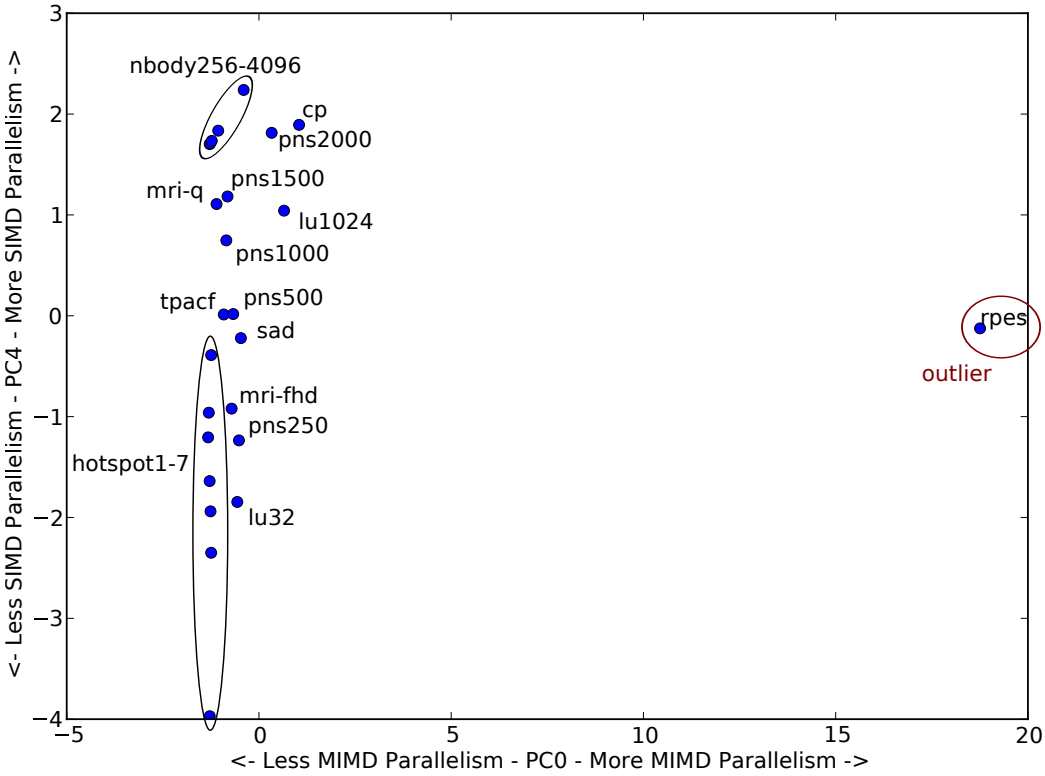

Figure 47: This plot compares MIMD to SIMD parallelism. It should be clear that these metrics are completely independent for this set of applications; the fact that an application can be easily mapped onto a SIMD processor says nothing about its suitability for a multi-core system. A complementary strategy may be necessary that considers both styles of parallelism when designing new applications.

collectively describe the number of instructions executed in each kernel. As described in the analytical model developed by Hong et. al. [67], these dynamic instruction counts are strong determinants of the total execution time of a program, and therefore the high degree of correlation is expected. What is not obvious is the relationship between the number of DMA calls executed before a kernel is launched and these instruction counts. We find that across all principal components, there is at least one metric that is available before launching a kernel that is highly correlated with the dynamic metrics in that component. We exploit this property in Section 8.5.4.5 to build a predictive model for application execution time using only static metrics.

PC2: Data dependences . We believe that the second principal component exposed the most significant and non-obvious relationship in this study. It indicates that data dependences are likely to be propagated throughout all levels of the programming model; if there is a large degree of data sharing between instructions, then there is likely to be a large degree of data sharing among threads in each CTA and among all CTAs in a program. Notice that this component shows that registers that are alive at context switch points, Memory Sharing, and total kernel count are highly correlated. Data is typically passed from one thread to another at context switch points. More context switch points imply more opportunities for sharing data between threads and more registers alive at these context switch points imply more data that can be transferred to another thread. Memory sharing measures exactly, the amount of memory that is passed from one thread to another through shared memory. Finally, CTAs cannot reliably exchange data within the same kernel, but kernels have implicit barriers between launches that allow data to be exchanged between CTAs in different kernels. The correlation between memory sharing and kernel count seems to indicate that programmers will break computations that are required to share data among CTAs into multiple kernels. Furthermore, it seems to indicate that programs are either embarrassingly parallel at all levels, from the instruction level up to the task level, or have dependences at all levels.

PC3: Memory Intensity . The next principal component is composed almost entirely of metrics that are associated with the memory behavior of a program. It should be clear that kernels that are given access to a large pool of memory via pointers are likely to access a significant amount of it. It is also interesting that applications that access a large amount of memory are likely to access it relatively efficiently, possibly because it can be accessed in a streaming rather than a random pattern. This component reveals that the memory intensive nature of applications

is reflected in all levels of the memory hierarchy, from the register pressure to the ratio of memory to compute instructions to the amount of memory accessible by a kernel; for this analysis, a program with high register pressure can be predicted to be very memory intensive. Finally, this component is negatively correlated with dynamic floating point instruction count, indicating that applications either stress the memory hierarchy or the floating point units in a given processor, but not both. This information could be used in the design of highly heterogeneous architectures where some processors are given low latency and high bandwidth memory links, others are given extra floating point units, and workloads are characterized and directed to one or the other accordingly.

PC4: Control Flow Uniformity/SIMD Parallelism . The final component exposes several very interesting relationships involving the Activity Factor of an application. Recall that Activity Factor refers to the average ratio of threads that are active during the execution of a given dynamic instruction. First, Activity Factor is directly correlated with special instructions, indicating that it is unlikely that texture or transcendental operations will be placed immediately after divergent branches; if a special instruction is executed by one thread, it is likely to be executed by all other threads. This relationship is reversed for double precision floating point instructions; programs that execute a significant number of double precision instructions are likely to be highly divergent.

Discussion . Though the intent of this analysis was to identify uncorrelated metrics that could be used as inputs to the regression model and cluster analysis in the following sections, PCA also exposed several key relationships between program characteristics that may inform the design of CUDA applications, data parallel compilers, or even new processors optimized for different classes of applications. For example,

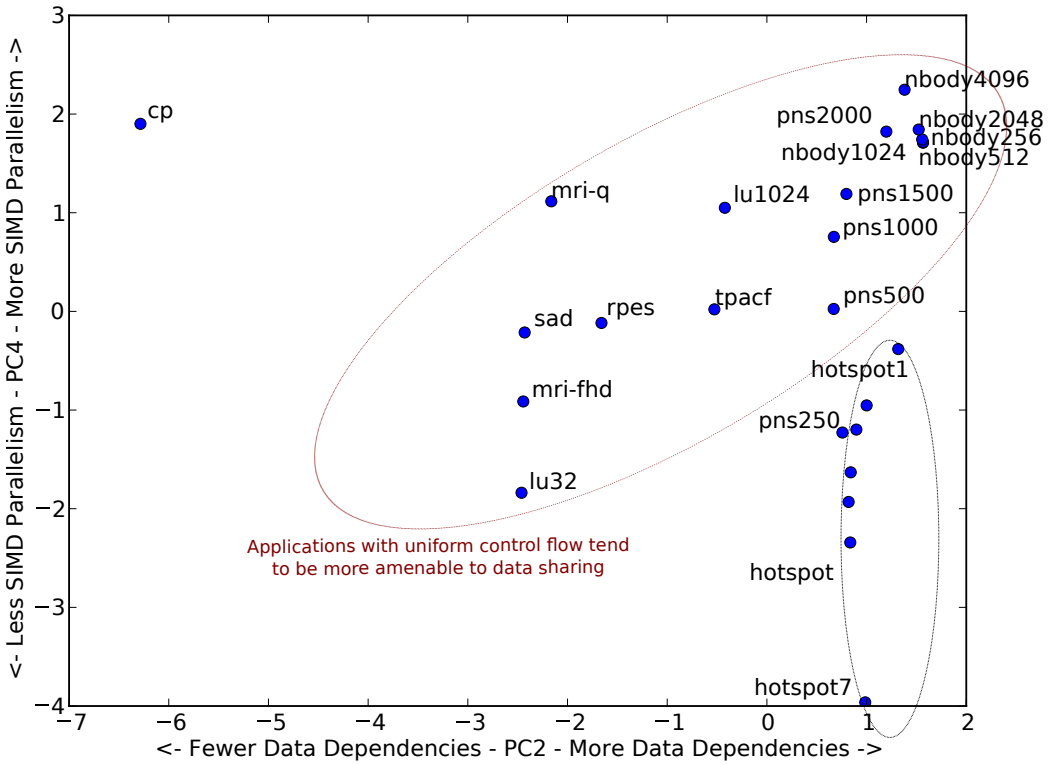

Figure 48: A comparison between Control Flow Divergence and Data dependences/Sharing. Excluding the hotspot applications, applications with more uniform control flow exhibit a greater degree of data sharing among threads. Well structured algorithms may be more scalable on GPU-like architectures that benefit from low control flow divergence and include mechanisms for fine grained inter-thread communication.

we expected the dynamic single precision floating point instruction count to be negatively correlated with the dynamic double precision count. However, as can be seen in Figure 46, they are not correlated all, indicating that many applications perform mixed precision computation. After examining several applications, we realized that some used floating point constants in expressions involving single precision numbers. The compiler interprets all floating point constants as double precision unless they are explicitly specified to be single precision, and any operations involving these constants would be cast up to double precision, performed at full precision, and then truncated and stored in single precision variables. This is probably not the intention of the developer, and in processors where there are limited double precision floating

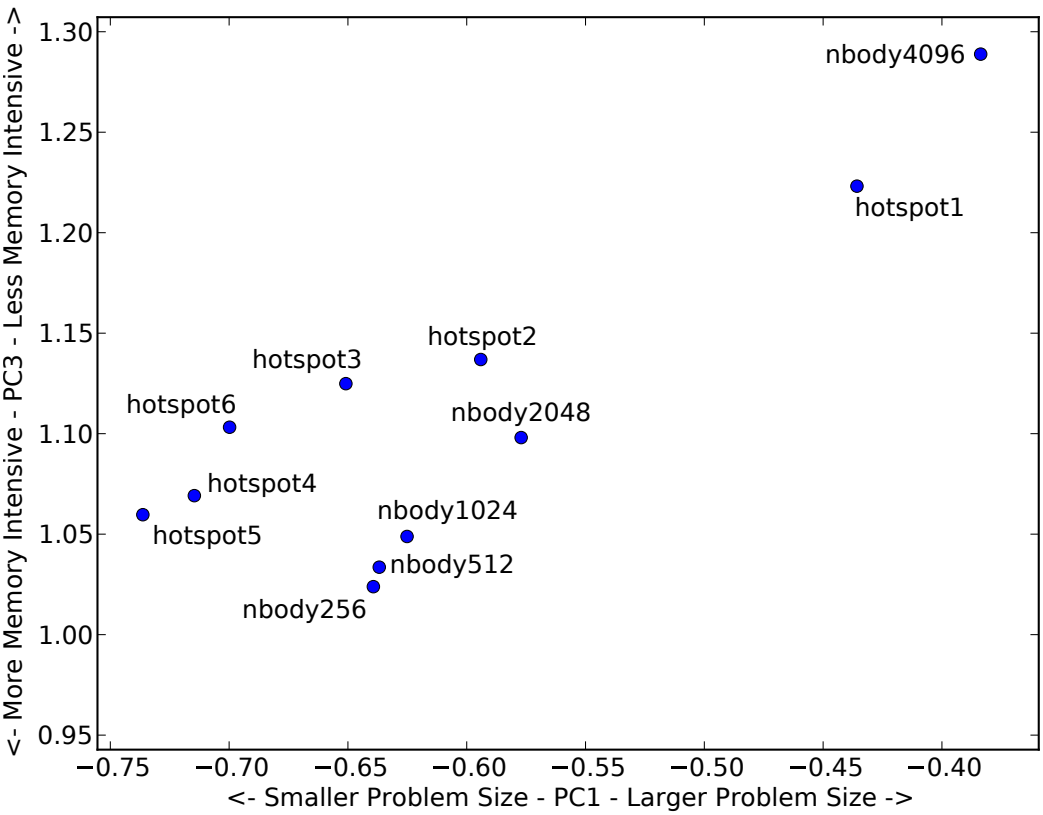

Figure 49: This figure shows the effect of increased problem size on the Memory Intensity of the Nbody and Hotspot applications. While this relationship probably will not hold in general, it demonstrates the usefulness of our methodology for characterizing the behavior of individual applications. We had originally expected these applications to become more memory intensive with an increased problem size; they actually become more compute intensive. This figure is also useful as a sanity check for our analysis, it correctly identifies the Nbody examples with higher body counts as having a larger problem size.

point units, such as the C1060, this may incur a significant performance overhead.

8.5.4.4 Cluster Analysis

Cluster Analysis is intended to identify groups applications with similar characteristics. It is useful in the development of benchmarks suites that are representative of a larger class of applications, visualizing application behavior, and in the context of this study, simplifying the development of accurate regression models via regression trees. For this analysis, we project the original application data onto the principal components. This allows the individual applications to be compared in terms of the

principal components, for example, we can say that *cp* has the least data dependences and *nbody4096* has the most. Though there are 10 possible projections of the five principal components, we present only the three most interesting examples. Figure 49 shows that MIMD and SIMD parallelism are not correlated, Figure 48 highlights a non-intuitive relationship between control flow uniformity and inter-thread data sharing, and Figure 47 illustrates how the *nbody* and *hotspot* applications scale with problem size.

8.5.4.5 Regression Modeling

The goal of this study is to derive accurate models for predicting the execution time of CUDA applications on heterogeneous processors. If possible we would also like to be able to make these predictions using only metrics that are available before a kernel is executed. As shown in Section 8.5.4.3, 85% of the variance across the set of metrics can be explained by five principal components, each of which is composed of at least one static metric that is available before the execution of a kernel. For example, according to the PCA, the size of DMA transfers, the number of DMA transfers, the number of live registers at context switches, the extent of memory accessible by each kernel, and the number of double precision instructions are all statically available metrics that should be good predictors of kernel performance.

In this example, we use the polynomial form of linear regression to determine a relationship between static program metrics and the total execution time of an application. Modeling M variables, each with an N-th order polynomial, requires at least $N * M$ samples for an exact solution using the least squares method for linear regression. This limits the degree of our polynomial model in cases where only a few samples are available, which may be a concern for models that are built at runtime as a program is executing.

Though linear regression will generate a model for predicting the execution time

of a given application on a particular CPU or GPU, it can generate predictions that are obviously not valid. For example, it is common for the model to predict short running kernels to have negative execution times, or predict that the execution time of a relatively more powerful processor is significantly slower than another processor that is invariantly slower in all other cases. In order to account for these cases, we applied a technique typically used in image processing, Projections Onto Convex Sets(POCS) [156], to each prediction. POCS works by applying a series of transformations to data, each of which enforces some a priori constraint on the data that must be true in all cases. If each successive transformation causes another invariant property to be violated, the iterative application of each transformation is necessary to find a result that satisfies all of the a-priori constraints. In this case, we impose the constraints that all execution times must be greater than 0, and that all predictions for a given application on a given architecture should be within two standard deviations of the mean of all execution times of the same application on other architectures.

In this study, we recorded the metrics described in Section 8.5.2 and execution times of 25 applications on seven different processors. We used this data to predict the execution times of new applications on the same processor and the same application on different processors. We found that the models are most accurate when predicting the same application on a different, but similar style of architecture. For example, predicting the execution time of an application on an 8800GTX GPU that has previously run on an 8600GS and a 280GTX. Predicting the execution time of a new application on the same processor is also relatively accurate. However, models that attempt to, for example, predict GPU performance given CPU training data are wildly inaccurate. In these cases we believe that it will be necessary to develop separate models for each cluster of applications using a technique akin to regression trees on the data presented in Section 8.5.4.4.

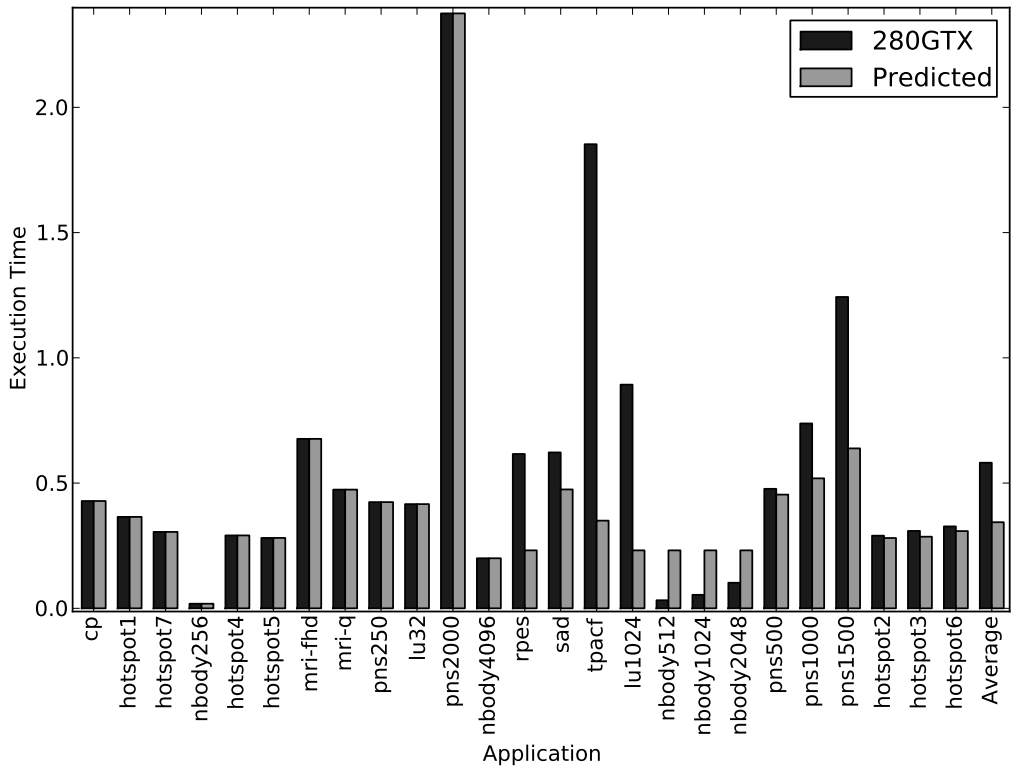

Figure 50: Predicted execution times for the 280GTX using only static data. The left 12 applications are used to train the model and the predictions are made for the rightmost 13 applications.

Application Modeling . Our first experiment attempts to build a model for the execution time of a the remaining 13 applications on an NVIDIA 280GTX GPU using 12 randomly selected applications for training. We chose to only include results from the 280GTX in this section because the models for the other architectures yielded similar results. Figure 50 compares the predicted execution time to the actual execution time for each of the 25 applications. Note that this model is intended to be used at runtime in a system that launches a large number of kernels, therefore it should be perfectly accurate for kernels that it has already encountered. This model is the most accurate for the hotspot, sad, and smaller sized PNS applications where all predictions are within 80% of the actual execution time. It is relatively inaccurate for the *nbody*, *tpacf*, *rpes*, *lu*, and larger *PNS* applications. In the worst case, the execution time of tpacf is predicted to be only 22% of the recorded time.

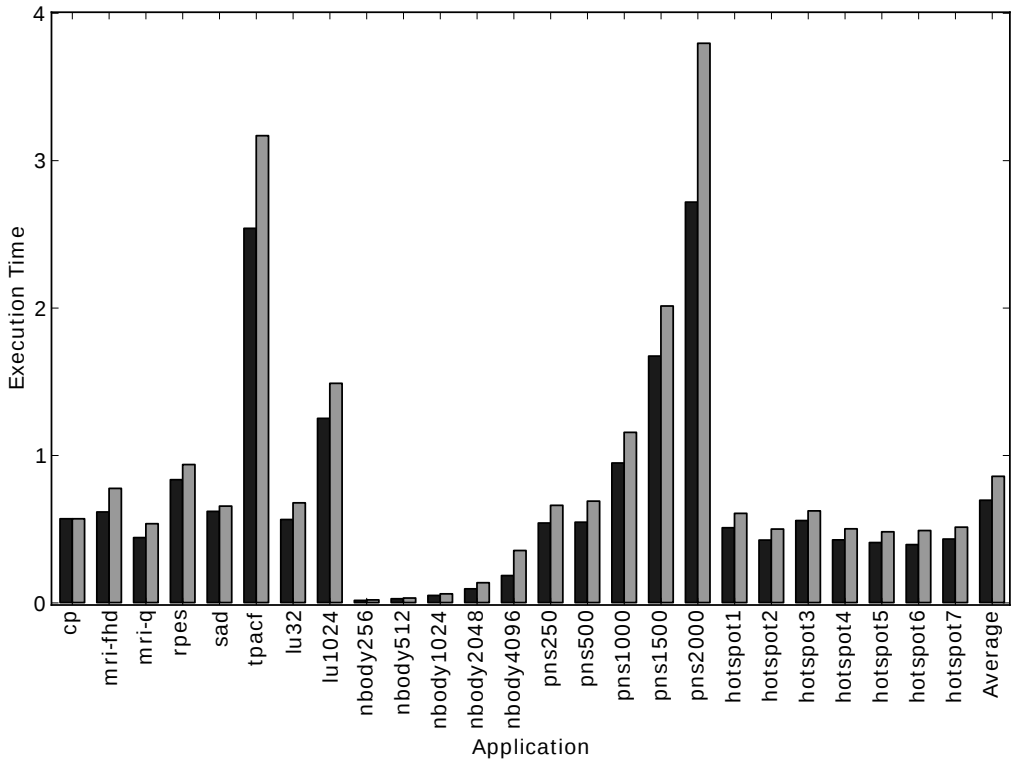

Figure 51: Predicted execution times for the 8800GTX using only static data and all other GPUs to train the model. Black indicates the measured time, and gray is the predicted time.

GPU Modeling . The next experiment uses the actual execution times of each application on the Geforce 280GTX, Geforce 8600GS, and Tesla C1060 to predict the execution time of the same applications on the Gefore 8800GTX. Figure 51 shows the predicted execution time as well as the measured execution time for each application. This model was the most accurate that we evaluated, the worst case being the *pns2000* application, for which to total execution time is predicted to be 3.9s and the actual execution time was 2.8s. In all other cases, the model underestimates the performance of the 8800GTX by between 16% and 1%. It is worthwhile to note that this model is able to predict the impact of increased problem size on the same application. For example, the execution time of each run of the *nbody* application is predicted to increase as the number of bodies simulated increases. The Parboil benchmark *tpacf* is relatively difficult to be predict by this model, and, in fact, the CPU and application

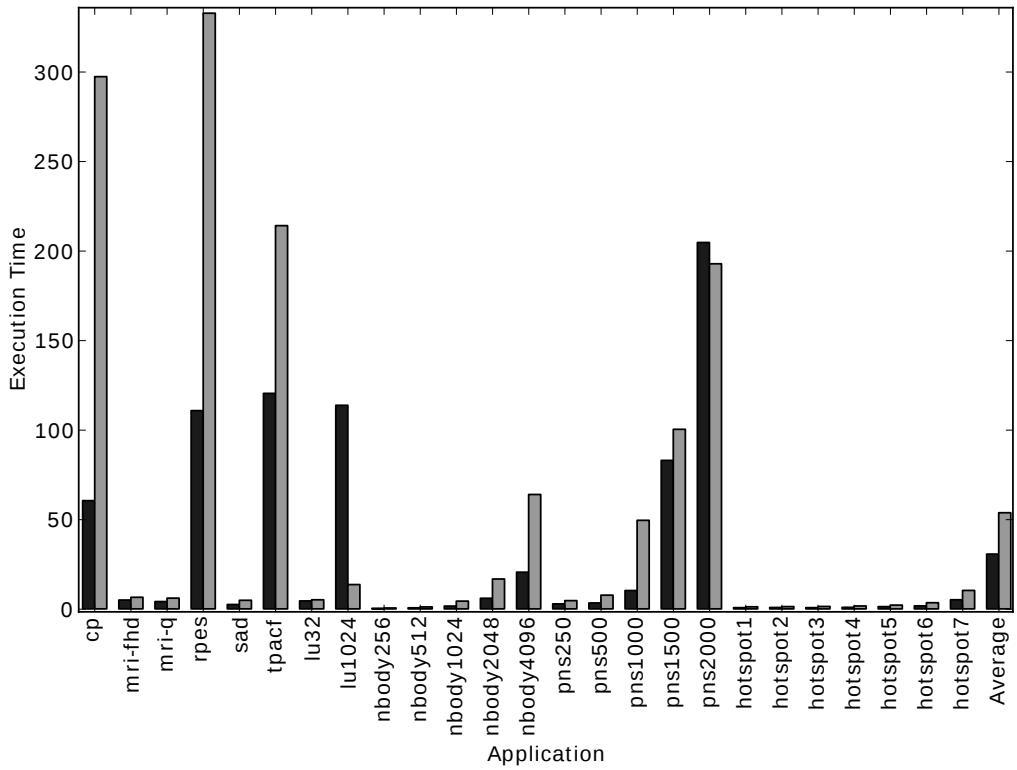

Figure 52: Predicted execution times for the AMD Phenom processor using the Atom and Nehalem chips for training.

models as well. This model always places the performance of the 8800GTX between that of the 8600GS and the Tesla C1060, which is also true for the actual execution times of all applications.

CPU Modeling . The third experiment uses training results from the Intel Nehalem and Intel Atom processors to predict the performance of the AMD Phenom processor. It should be immediately clear that moving to CPU platforms changes the relative speed of each application. For example, on all of the GPU processors, the performance of *sad* and *rpes* are relatively similar. However, on the CPUs, *sad* is nearly 25x faster than *rpes*. This reinforces the point that GPUs and CPUs are more efficient for certain classes of applications than others even when they are both starting with the same implementation of the program.

Compared to the GPU model, the CPU model is slightly less accurate as can

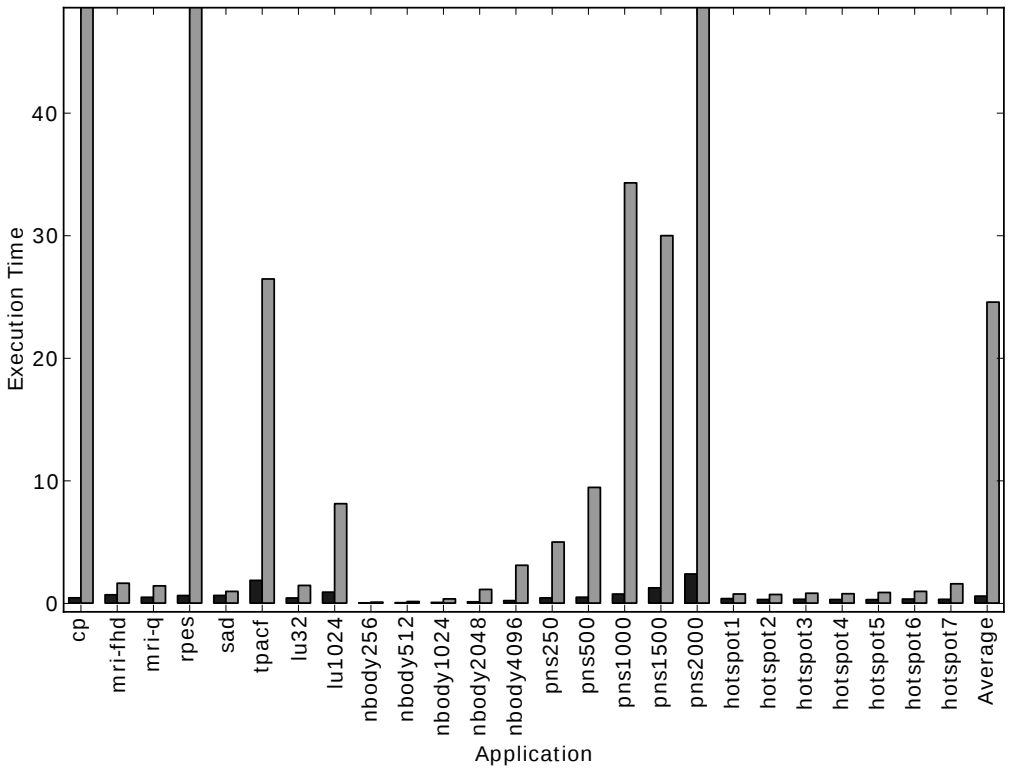

Figure 53: Predicted execution times for the 280GTX using all of the other processors for training. This is the least accurate model; it demonstrates the need for separate models for GPU and CPU architectures.

be seen by comparing Figure 51 and Figure 52. The CPU model in Figure 52 is the most accurate for *hotspot*, the larger *pns* benchmarks, the *mri* benchmarks, and *nbody*. For those applications the predicted execution times fall with 80% of the measured execution times. The model is the least accurate for the *cp* application, which is predicted to take 62s and actually takes 294s to execute. It is interesting to note that this application only takes 54s on the Intel Nehalem processor, which is typically competitive with the AMD Phenom. Whatever machine characteristic causes this large discrepancy in performance is not captured in our set of machine metrics. It is possible that a more detailed model including more machine metrics would be able to capture that relationship.

GPU-CPU Modeling . The final experiment demonstrates a case in which our methodology fails to generate an accurate model. Figure 53 presents the predictions for a model for the 280GTX using results from all other processors for training. This model excessively overestimates the execution time of each application with only *cp*, *hotspot*, and *sad* being within 50% of the total execution time. As our cluster analysis shows, GPU and CPU style architectures have very different machine parameters and combining them in the same model significantly reduces the accuracy of the model. It motivates the need for a two stage modeling approach in which applications and processors are first classified into related categories with similar characteristics and then modeled separately.

8.5.4.6 Discussion

A significant result of this paper is that the methodology of principal components analysis, cluster analysis, and regression modeling is able to generate predictive models for CPUs and GPUs, suggesting that there are certain characteristics that make an application more or less suitable for a given style of architecture. Unfortunately, while the regression method used in this study can generate an accurate model, it usually includes complex non-linear relationships that are difficult to draw any fundamental insights from. Additional analysis is needed to discover these relationships, perhaps by determining correlations between application metrics and relative performance on a particular system in future work.

Though the relationships described in the preceding sections are applicable only for the applications machine configurations used in this study, the methodology is a universally valid tool that can be applied to any set of applications. An extension of this work would be to increase the set of possible benchmark applications and input sizes then use PCA and cluster analysis to select the most representative applications and inputs in a manner described in detail in [123].

Finally, this study exposed several non-intuitive relationships between application characteristics, for example, that applications with highly uniform control flow are more amenable to fine-grained synchronization and inter-thread communication. Discovering these relationships is becoming increasingly important as they can expose opportunities for architecture optimizations and influence the selection of well structured algorithms during application development. Clearly, there is a pressing need for further analysis and additional data.

8.5.5 Concluding Remarks

This section presents an emulation and translation infrastructure and its use in the characterization of GPU workloads. In particular, standard data analysis techniques are employed to characterize benchmarks, their relationships to machine and application parameters, and construct predictive models for choosing between CPU or GPU implementations of kernel based on Ocelot's translation infrastructure. Accompanying the insights this approach provides is a clear need for deeper and more refined characterizations and prediction models - a subject of ongoing and future work.

8.6 A High Level Language Frontend for Harmony

This case study describes the design of a programming language front-end targeting the Harmony execution model. An application is specified in Datalog$_{LB}$, a declarative programming model for database and business analytics applications. It is then passed through a series of compilation stages that progressively lower datalog into Relational Algebra (RA) operators, RA operators into Harmony kernels, and finally generate PTX for each kernel. A set of algorithm skeletons for each of the RA operators is provided as a CUDA template library to the compiler. During compilation, the skeletons are instantiated to match the data structure format, types, and low level operations used by a specific RA operator. The final Harmony application is serialized into a binary format that is executed by an implementation of the runtime, such as the prototype described in Section 8.2.

8.6.1 Datalog$_{LB}$

The LogicBlox platform consists of the Datalog^{LB} language, and a runtime for evaluating Datalog^{LB} programs over large sets of data. Datalog^{LB} is a variant of Datalog [1], with extensions to support the development of an entire enterprise application stack: from user-interface controls, to workflow involving complex business rules, to sophisticated analysis over data.

The declarative nature of Datalog^{LB} is the key reason for its suitability for rapid application development. Datalog^{LB} is a logic programming language, where computations over data are expressed in terms of logical relations among sets of data: e.g., conjunctions, implications, etc. Datalog^{LB} also provides features for extra-logical operations, such as active rules for updating state, and explicit sequencing of rules or groups of rules. However, these features are carefully designed such that programmers only have a limited number of ways to change state, or construct control flow. Compared to popular imperative programming languages such as Java or C++, Datalog^{LB}

abstracts away much detail about the actual execution of a program from application developers: developers only specify logical relationships between data; the Datalog^{LB} compiler constructs a suitable execution plan by extracting data and control flow from the Datalog^{LB}program. Compared to emerging distributed programming languages for GPUs such as Map-Reduce [64], Datalog^{LB} expresses a richer set of relational and database operations that are cumbersome to implement in Map-Reduce.

The abstraction and declarative nature that helps programmers reason about Datalog^{LB} programs are exactly what can potentially help create high performaning evaluation plans for Datalog^{LB} programs, as well. A Datalog^{LB} program's data flow is made explicit through the logical relations used to define data. Furthermore, well-defined properties associated with logical relations, such as commutativity and associativity of conjunctions, readily expose data parallelism in Datalog^{LB} programs. Lastly, the limited number of control operators (e.g., one sequencing operator, no unlimited recursion), makes it easy to construct a finite data and control flow graph. The current Datalog^{LB} compiler decomposes a program into such a data-control flow graph, which then serves as an evaluation plan for the LogicBlox runtime. At the time of this writing, the Datalog^{LB} compiler and the runtime do not, however, take advantage of the readily exposed data, task, and pipeline parallelism in a Datalog^{LB} program. Furthermore, the compiler/runtime do not take advantage of the different hardware available on different deployment environments. Lastly, the Datalog^{LB} compiler does not have the ability to determine which part of the computation can be best executed where, given the available hardware.

Since LogicBlox applications target the cloud where hardware per instance of deployment can vary, it is crucial for LogicBlox to be able to take advantage of the hardware available. The Redfox compiler replaces the existing LogicBlox compiler and decomposes the program into a graph of Harmony kernels. In this representation the graph exposes task and pipeline parallelism that can be exploited by the Harmony

Set Intersection:	$D \leftarrow A \cup B$
Set Union:	$D \leftarrow A \cap B$
Set Difference:	$D \leftarrow \{x \in B \mid x \notin A\}$
Projection:	$D \leftarrow \pi_{a_1,\ldots,a_n}(A)$
Selection:	$D \leftarrow \sigma_{a\theta b}(A)$
Cross Product:	$D \leftarrow A \times B$
Join:	$D \leftarrow A \bowtie B$

Table 9: The set of relational algebra operations.

runtime, and the data parallel representation of kernels allows them to be effectively mapped onto highly parallel processors such as GPUs or multi-core Central Processing Unit (CPU)s.

8.6.2 Relational Algebra Operators

RA consists of a set of fundamental transformations that are applied to sets of primitive elements. Primitive elements consist of n-ary tuples that map attributes (or dimensions) to values. Each attribute consists of a finite set of possible values and an n-ary tuple is a list of n values, one for each attribute. Another way to think about tuples is as coordinates in an n-dimensional space. An unordered set of tuples of the type specifies a region in this n-dimensional space and is termed a 'relation'. Each transformation included in RA performs an operation on a relation, producing a new relation. Many operators divide the tuple attributes into key attributes and value attributes. In these operations, the key attributes are considered by the operator and the value attributes are treated as payload data that is not considered by the operation. A relational algebra application is specified as a dataflow graph of operators, making for a natural representation in the Harmony execution model by mapping operators to kernels and relations to variables. The complete set of RA operators is listed in Table 9. They are described in detail as follows:

Set Intersection Set intersection is a binary operator that consumes two relations to produce a new relation consisting of tuples with keys that are present in both of the input relations.

Set Union Set union is a binary operator that consumed two relations to produce a new relation consisting of tuples with keys that are present in at least one of the input relations.

Set Difference Set difference is a binary operator that consumes two relations to produce a new relation of tuples with keys that exist in one input relation and do not exist in the other input relation.

Projection Projection is a unary operator that consumes one input relation to produce a new output relation. The output relation is formed from tuples from the input relation after removing a specific set of attributes.

Selection Selection is a unary operator that consumes one input relation to produce a new output relation that consists of the set of tuples that satisfy a predicate equation. This equation is specified as a series of comparison operations on tuple attributes, for example, that the value of the first attribute is less than 5.

Cross Product Cross product is a binary operator that combines the attribute spaces of two relations to produce a new relation with tuples forming the set of all possible ordered sequences of attribute values from the input relations.

8.6.3 Algorithm Skeletons

The RA operators are implemented using various algorithm skeletons that allow the same high level algorithm to be readily adapted to operations over complex data types. This approach is commonly used in compilers for high level domain specific

languages such as Copperhead [24], Optix [121], or even in generic languages such as Haskell [93].

Relations are stored as a densely packed array of tuples that is sorting using a comparison function that operates on tuple attributes and defines a strict weak-ordering over all tuples in the relation. The sorted form allows for efficient array partitioning and tuple lookup operations. The algorithm skeletons are designed to simultaneously maintain this sorted property on relations as well as achieve good performance on multi-core processors. Additionally, several of the RA operators are similar enough to be implemented with a single skeleton. This is mainly applicable for the set family of operators. The individual algorithms are discussed in detail next:

Partition, Merge, Gather The first algorithm skeleton is used to implement the set family of RA operators. It is broken out into three major stages, 1) the input relations are partitioned into independent sections that can be processed in parallel, 2) the sections are merged together using a specialized function for each RA operator generating independent sets of results, and 3) the individual sections are gathered together back into a dense sorted array of tuples. Each stage of the algorithm requires a global synchronization operation and intermediate results to be stored to a shared memory region. In CUDA, this currently forces the results to be written out to global memory and the implementation to be partitioned into three independent kernels. So, the final Harmony IR representation of these operators will actually consist of a series of kernels that operate on a shared set of variables representing the relations.

Each of these stages is expected to be memory bound on the target processors. The initial partitioning stage is done in-place and the partitions are sized such that the partitioning does not consume a significant portion of the total execution time. The sorted property of the input relations is exploited to perform a recursive double sided binary search that progressively places more restrictive bounds on the input relations,

creating sections that may contain tuples with overlapping keys. The double sided binary search is a parallel implementation of the algorithm presented by Baeza-Yates et al [9]. It works by partitioning one of the input relations into N sections bounded by pivot elements, then using a binary search to lookup the tuples in the other input corresponding to the pivots creating a series of partitions with overlapping ranges of tuples. This process can be repeated recursively by swapping the inputs, and subdividing the individual partitions using the same method. In this implementation, the process is repeated until the partitions have been reduced to a predetermined size. The initial stage partitions the arrays into one partition for each Cooperative Thread Array (CTA), and the recursive stages are handled within each CTA, allowing for maximum concurrency.

The merge operation is the most complicated of the three stages. Once the inputs have been partitioned, each pair of partitions is assigned to a separate PTX CTA to be processed independently. This stage of the algorithm implements a merge operation that is commonly used in CUDA implementations of sorting [25, 104, 129]. It requires at least one load operation for each of the tuples in the input partitions and one store for each generated result. The merge operation is implemented by scanning one of the input partitions one chunk of tuples at a time that can be processed by a fixed number of threads in a CTA. This chunk is loaded into on-chip shared memory or registers for fast access. A corresponding chunk from the second input partition is also loaded loaded into shared memory and compared against tuples in the first chunk. The exact comparison function that is used depends on the RA operator being performed, for example, set intersection would check for matching tuples in each chunk whereas set difference would check for the presence of a tuple in one chunk but not the other. Chunks from the second input partition are scanned until they go out of range of the first chunk, at which time a new chunk is loaded from the first partition and the process is repeated. When matches are found, they are gathered into shared

memory until a threshold is reached and eventually written out to a preallocated temporary array. The chunk copies into and out of shared memory are carefully designed such that they maximize DRAM bandwidth. Additionally, the comparison operations among chunk elements are cooperatively performed by multiple threads and hand scheduled by unrolling loops and manually inserting barriers to eliminate sequential instruction dependences. An empircal measurements using performance counters show that the algorithm is bottlenecked by the transfers in to and out of shared memory.

The final gather stage requires first computing the position of each section of the result in the final array. This is performed by updating a histogram during the merge stage, followed by an out-of-place scan operation over the histogram buckets, a common CUDA programming pattern [102]. Again, the number of partitions is sized such that this operation is relatively inexpensive compared to the merge phase. Once the position of each section in the output relation is determined, elements need to be copied from a temporary buffer for each section into the final array using at least one load and one store operation for each element.

Remove Keys, Remove Duplicates The next skeleton implements the select RA operator. It is relatively simple compared to the previous skeleton: the first stage only requires a transform pass over tuples in the array to decompress the tuple attributes, remove some attributes, and then compress the tuple with the reduced set of attributes. This is a completely data-parallel operation that can be trivially partitioned among a large number of threads and CTAs. However, it may produce duplicate tuples that were originally distinguished by attribute values that no longer exist. These duplicates can be removed efficiently using the sorted property of the data structure since identical tuples will be stored consecutively. This reduces to a stream compaction problem, and the form of the algorithm skeleton follows the

approached used in [14].

Selective Gather This skeleton is used to implement the selection RA operator. Each tuple is examined and its key is compared using a user-defined predicate function. If the comparison succeeds, the tuple needs to be copied into the output relation, otherwise it is discarded. The algorithm is implemented as two passes over the relation. The first pass records the number of tuples that will be copied by each CTA, storing this value in a histogram entry. An out-of-place prefix sum determines the position that each CTA should begin writing in the output relation. The buckets in the histogram are sized appropriately to make this an insignificant portion of the total algorithm execution time. The final stage scans the input relation again, this time gathering the tuples together that satisfy the comparison function and writing them into the output array. The tuples are buffered in CTA shared memory and written out as bulk transfers to improve memory efficiency.

It is worth noting that the redundant first pass could be eliminated by using a more flexible data structure format that allowed for efficient parallel resize operations to take place as new tuples that satisfy the comparison function are discovered. This is left as future work in this case study.

Expand The expand skeleton is used to implement the cross product RA operator. This is probably the simplest operation. The number of tuples in the output relation is the product of the number of tuples in the input relations, and computing this size does not require a separate pass. Therefore, the algorithm is implemented using a single pass that iterates over each combination of elements in each of the input relations and invokes a user defined function to combine them. The combined tuple is written out to the output relation. No buffering stage is required because the results are generated as contiguous, densely packed chunks.

Partition, Merge-Expand, Gather The final algorithm skeleton is used to implement the join RA operator. It is easily the most complicated algorithm, sharing many characteristics with the (Partition, Merge, Gather) skeleton used for the set operations. However, it is further complicated by the second stage of the algorithm, which involves identifying subsets of the partitioned relations with overlapping attributes and performing the cross product for each subset. This presents a significant problem to parallel implementations of the algorithm that eventually write to a statically allocated, dense array. Namely, that the number sections and tuples in each section of the output relation is not known until very late in the computation. In the one extreme case, there could be a single section with matching attribute values containing all tuples in each array, in which case the join would become a cross product. In another extreme case, sections could contain at most one tuple each, in which case the join would become a set intersection.

This problem is partially addressed by the initial partitioning stage. Sections from the input relations must always fall into the same partition, so a pair of input relations, that have been partitioned into M partitions (each of size N) may only produce as many as $M * N^2$ tuples in the output relation in the worst case, rather than $(M * N)^2$ in the general case. However, temporary storage still needs to be allocated for N^2 possible tuples rather than N tuples in the set intersection case, limiting the number of partitions that can be processed independently.

8.6.4 CUDA Kernel Skeleton Specialization

Once a relational algebra operator has been mapped to a skeleton, the skeleton is instantiated for the data types of the relation, and the low level operation performed in the case of selection or projection. Skeletons are CUDA implementations of RA operators that are templated on the tuple type and possibly the operation type as well.

```
template<typename ResultTuple, typename LeftTuple, typename RightTuple>
__device__ void product(typename ResultTuple::BasicType* result,
    const typename LeftTuple::BasicType* leftBegin,
    const typename LeftTuple::BasicType* leftEnd,
    const typename RightTuple::BasicType* rightBegin,
    const typename RightTuple::BasicType* rightEnd)
{
    uint threads = blockDim.x * gridDim.x;
    uint id      = threadIdx.x + blockDim.x * blockIdx.x;

    uint leftSize   = leftEnd  - leftBegin;
    uint rightSize  = rightEnd - rightBegin;
    uint resultSize = leftSize * rightSize;

    for(uint i = id; i < resultSize; i += threads)
    {
        uint leftIndex  = i / leftSize;
        uint rightIndex = i % leftSize;

        result[i] = tuple::combine<LeftTuple, RightTuple, ResultTuple, 0>(
            leftBegin[leftIndex], rightBegin[rightIndex]);
    }
}
```

Figure 54: The skeleton CUDA implementation of cross product.

Cross Product Skeleton Figure 54 shows the CUDA skeleton for the cross product relational algebra operator. This is one of the simplest operations. Note that the function is templated on the tuple types of the two input relations as well as the output relation. A blocked set of threads iterate over each combination of elements in the input relations and use a template combination function to generate the associated element in the output relation. The tuple data storage format is defined by the Tuple class, which uses a form of compression to pack multiple attributes into single words of data so that they can be accessed efficiently as a single memory transaction by a single thread.

8.6.5 The RedFox Compiler

The compiler itself is broken out into several modules that perform different functions; these modules and their interactions are depicted in Figure 55. Collectively they lower Datalog^{LB} source files into functionally equivalent Harmony IR binaries of PTX kernels. These binaries can be executed by an implementation of the Harmony

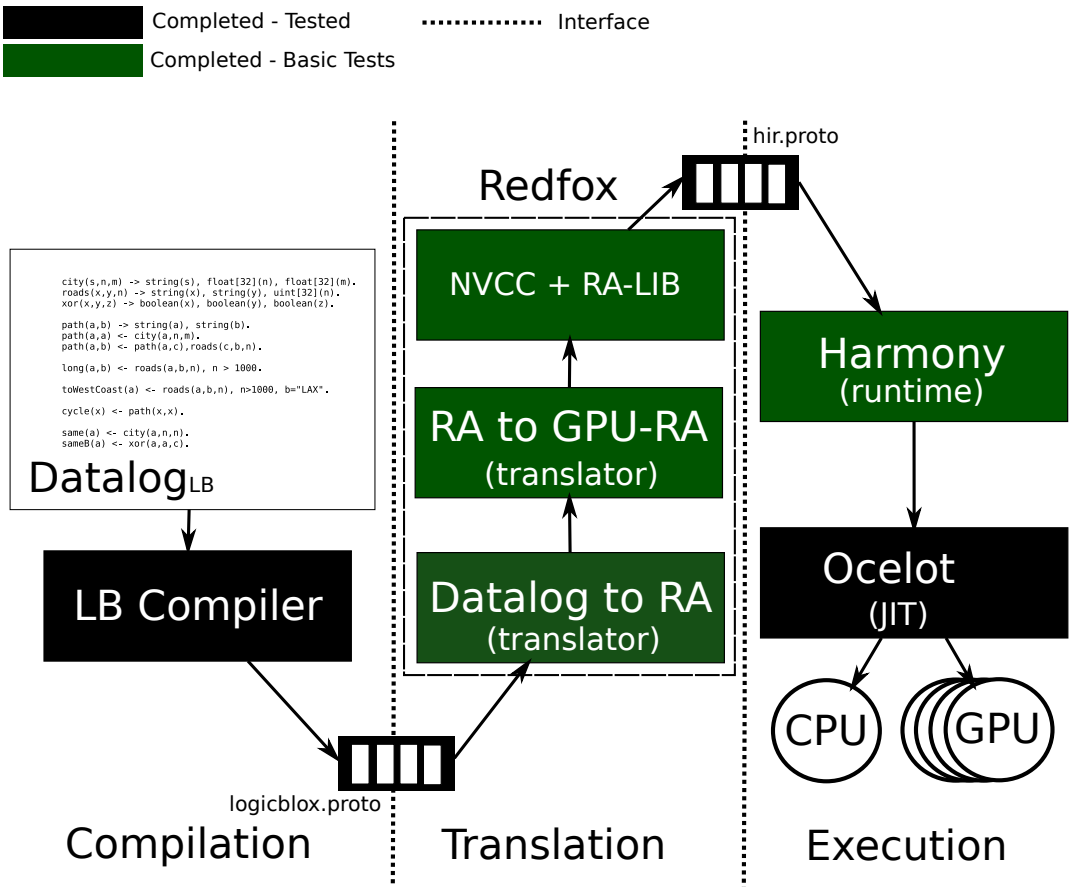

Figure 55: The major components in the RedFox compiler. The Datalog^{LB} language frontend converts programs into an AST representation. Datalog^{LB} operations are converted into a series of RA operations. These are converted one at a time into equivalent PTX kernels that are embedded in a Harmony binary and executed by the Harmony runtime and Ocelot dynamic compiler.

runtime that schedules the PTX kernels onto CPUs or GPUs.

The Datalog Frontend The LogicBlox datalog frontend parses a Datalog^{LB} source file into an Abstract Syntax Tree (AST) that stores information about relations and language clauses that operate on them. Clauses that contain multiple compound operations are broken down into atomic operations that can be executed individually. Eventually this AST is lowered into an execution plan. A list of all relations and their associated types is stored in this representation along with a data-flow graph of operations. The data-flow graph is augmented with the concept of a fixed-point

graph with circular dependences that executes until there is no change in any of the circular data dependences. This is necessary to implement recursive operations in Datalog^{LB} . This frontend is used both for the Redfox compiler and a CPU only compilation chain for Datalog^{LB} . The remaining components of the compiler are specific to Redfox.

The Datalog to RA Translator The next component in the compiler translates from Datalog^{LB} into RA operators. It performs a simple mapping from each Datalog^{LB} atomic operation to a series of abstract RA operators. Relations are also translated into equivalent types. The structure of the data-flow graph is left intact without any modification.

The Harmony Execution Plan Up to this point, the program is still specified purely as a data-flow graph with fixed-point sections. This is not directly compatible with the control-flow-graph of basic blocks representation of Harmony programs. The data-flow graph is topologically sorted to create an execution plan for a Harmony application, creating a sequential ordering of RA operators that satisfies all data dependencies. Fixed-point sections are converted into loops over the section and checks for changes in the output relations after each iteration. After this transformation is the program is expressed as a control-flow-graph of abstract RA operators.

The PTX Code Generator The final stage in the compilation process converts relations into Harmony variables and generates a series of PTX kernels for each RA operator. An CUDA skeleton is chosen for each kernel and the skeleton parameters are filled in from the operator type and the types of the input/output relations. The completed CUDA implementation is passed to NVCC, which compiles it into a PTX kernel. These PTX kernels are serialized one at a time into the Harmony binary.

Collectively, these components implement a complete compilation chain from a

Operator	Throughput
inner-join	26.4-32.3 GB/s
select	66.4 GB/s
set operators	45.8 GB/s
projection	54.3 GB/s
cross product	98.8 GB/s

Table 10: The throughput of each of the algorithm skeletons over random datasets. These operations are assumed to be memory bound, and throughput is counted as the size of the input relations plus that of the output relation divided by the total execution time of the algorithm.

CPU	AMD Phenom 9550 Quad-Core 2.2Ghz
GPU	NVIDIA Tesla C2050
Memory	8GB DDR-1333 DRAM
CPU Compiler	GCC-4.5.0
GPU Compiler	NVCC-3.2

Table 11: The microbenchmark test system configuration. All of the benchmarks are run only on the GPU.

Datalog^{LB} source file to a Harmony binary.

8.6.6 Microbenchmarks

This section covers a low level performance evaluation of this system by benchmarking and analyzing the performance of each of the skeleton algorithms. The tuple attributes are compressed into 32-bit integers. The compiler is able to change the word size used to store each tuple in cases where more attributes are required. However, tuples from many common applications can fit into 32-bit or 64-bit words. The tuples attribute values were generated randomly, and each relation contains approximately 16 million unique tuples. The system configuration is given in Table 11. These algorithms are memory bound on the GPU used in the test system, and the results are presented in achieved bandwidth. The theoretical peak memory bandwidth is 144GB/s, the achieved bandwidth using a simple copy benchmark is 102 GB/s.

Table 10 shows the throughput of each of the RA skeleton algorithms. Cross product achieves the best performance at 96.7% of the simple copy benchmark. As

set intersection
set union
projection
selection
selection (compound)
cross product
inner join
selection → join
selection (compound) → join

Table 12: Simple Datalog^{LB} benchmarks.

```
relationOne -> string(x).
relationTwo -> string(x), string(y).

relationResult(x,y) <- relationOne(x),relationTwo(x,y), x!="invalid".
```

Figure 56: An example of the Datalog^{LB} selection → join benchmark.

the algorithms become more complicated, the performance reduces, generally because of less regular memory accesses or multi-stage algorithms that require accessing each tuple multiple times. Selection achieves the next best performance, as it simply requires a series of linear scans over the input relations that can be performed at close to peak bandwidth. The scatters also occur linearly within a CTA, leading to highly efficient operations. Projection performs similarly, mainly being bottlenecked by the multi-stage algorithm that requires eliminating duplicate elements. The set operators require three intermediate stages and one extra load and store operation, reducing the possible bandwidth from 102GB/s to about 61 GB/s. The remaining inefficiencies probably arise in the second stage, which may not be completely memory bound. The performance of join is highly sensitive to the data distribution, ranging from between 26.4 GB/s and 32.3 GB/s.

8.6.7 Simple Application Performance

Moving on from microbenchmarks to complete Datalog^{LB} applications, several applications were written by hand that stress the performance of common Datalog^{LB}

clauses. These include compound operations such as selections with multiple conditions, and selections that are folded into joins. The complete list of benchmarks is listed in table 12.

Consider the specific example shown in Figure 56.

8.6.8 Insights and Discussion

8.6.9 Summary

This case study presents the design of a compiler framework for a high level declarative language commonly used for database and business analytics applications. The language is progressively parsed and lowered through a series of relational algebra representations that eventually are mapped to PTX kernels embedded in a Harmony IR binary that is executed by an implementation of the Harmony runtime on a system composed of heterogeneous accelerator boards. The design of the algorithms used to implement the RA operators is discussed at length, as well as their performance characteristics. This study stands as an example of the practical application of the Harmony execution model to a commodity system and application domain.

CHAPTER IX

MODEL EXTENSIONS AND FUTURE WORK

In this thesis, the Harmony execution model has been presented as a set of abstractions that can be used to express applications from multiple domains at different levels of granularity. The abstractions of kernels, control decisions, and variables can be applied at the level of instructions, accelerator kernels, or tasks in a distributed system. They also enable a generic set of static and dynamic transformations that may be used to optimize various aspects of applications. A structured data-parallel execution model is also presented at the kernel-level as a means for scaling the performance of individual kernels, and enabling portability of the same kernel across heterogeneous processors.

The final chapter of this dissertation summarizes the aspects execution model that have been presented thus far, and identifies several related open problems that merit additional work in the future.

9.1 The Harmony Execution Model

A major contribution of this dissertation is the introduction of the Harmony execution model. Harmony draws heavily from concepts and optimizations used in processor micro-architecture to extract a core set of abstractions and their execution semantics that can be applied to applications at different levels of granularity, and mapped to processors with heterogeneous characteristics.

9.1.1 Model Abstractions

Harmony expresses programs in terms of kernels, control decisions, and variables. Kernels are atomic, side-effect-free functions with explicit input and output parameters. Control decisions change the sequential flow of execution through a Harmony program, and variables are opaque wrappers around unformatted data. Kernels can internally perform any operation using any execution model. However, there needs to be a mechanism for executing kernels on some subset of processors in the system.

Harmony programs are executed by a runtime component on a system that is described using an abstract machine model in terms of processing elements, memory regions, and a central control unit. The central control unit executes the runtime component of Harmony, and directs the execution of kernels as well as the management and transfer of variables. The processing elements may have heterogeneous performance characteristics and execution models. The runtime is responsible for directing kernels to processing elements that can execute them, or performing an execution model translation that allows a kernel to be executed on a different type of processing element. Memory regions store Harmony variables, and have direct connections to processing elements that can access them. The runtime component of Harmony is responsible for copying variables into the memory region of the processor executing a kernel that lists them as parameters.

9.1.2 Applying Harmony At Different Scales

The Harmony execution model provides semantics and abstractions that can be applied at multiple scales. At the lowest level, Harmony can be used to describe the execution of a Von Neumann style application by a super-scalar or OOO processor. In this case, instructions are equivalent to kernels and registers are equivalent to variables. Harmony can also be applied to more coarse grained systems such as many-core system on chip processors, modular nodes composed of directly connected accelerator

boards, or complete nodes in a distributed system. At each of these levels, Harmony delegates the responsibility of scheduling compute intensive kernels onto heterogeneous processing elements to a runtime layer, which provides application portability across multiple generations or configurations of hardware with different numbers and types of processing elements.

9.1.3 Optimizing the Model

Various optimizations can be applied to the model statically or dynamically during execution. A fundamental optimization involves scheduling kernels on processing elements using historical and predictive performance models. Other optimizations include i) dynamically trading off parallelism for memory footprint by eliminating kernel-dependencies with variable renaming, ii) fusing kernels together into atomic units to reduce runtime overheads, and iii) speculatively executing kernels using branch or value prediction.

9.2 Important Directions for Future Work

There are also several interesting avenues for future work on related topics.

9.2.1 A Model of Locality

The scheduling algorithms used in the Harmony runtime evaluated in this dissertation only consider the data transfer overheads associated with moving variables between memory regions when scheduling kernels. This is purely a local decision that is made on a per-kernel basis. As memory systems are designed with less uniform latency and bandwidth characteristics, there will be a need for explicitly partitioning variables and distributing them across different levels of a memory hierarchy. A more sophisticated kernel scheduling algorithm that looked at the memory transfer overheads of a series of kernels in more detail could be a potential direction for future work. However, as this trend continues to multi-core processors, it may be necessary

to augment kernel execution models like PTX with a notion of locality and data distributions similar to Seqouia [50], Hierarchical Place Trees [155], or Chapel data distributions [40]. The important problem for these systems may be the distribution of data structures throughout the memory system coupled to the mapping of threads and thread groups to processing elements. Many data parallel applications implicitly assume data distributions (into arrays or grids) that enhance locality by partitioning data structures and distributing them to tasks or threads that process subsections in parallel. However, this partitioning information is typically lost after the application is expressed in a programming language. Structured and explicitly parallel kernel execution models will require these distributions and mappings to be made explicit.

9.2.2 Kernel Specialization

The concept of a kernel provides boundaries for providing different implementations of the same kernel with the same functional characteristics, but different performance characteristics. This idea was explored in this thesis in the context of different optimizations that are applied to a kernel depending on the processing element that it is scheduled on, but it could be taken much further than this. Different implementations of the same kernel using different high level algorithms could be supplied similar to N-version applications [30]. In this case, the runtime could choose from one of several algorithms, each of which may perform differently on different processors. Another, less extreme, approach could involve providing a built-in auto-tuning framework for kernels. This could be accomplished by specifying a set of kernel parameters as 'tunable' with valid ranges for tuning. Invoking the same kernel with different values for these parameters would perform the same function, but would use a different variation of the same algorithm. For example, by choosing a different blocking factor for matrix multiplication, or processing a different number of elements per thread.

9.2.3 Beyond Subkernel Formation

Subkernel formation described in Chapter 8 describes one technique that can be used to ease the problem of mapping an explicitly parallel kernel execution model onto different parallel processors. The proposed solution involves breaking a program into regions that are lazily translated and optimized by a dynamic compiler as groups of threads enter them. A natural extension would involve the compiler itself being implemented as a data-parallel kernel that could execute efficiently on the same processor, rather than locking subkernels and passing control back to a sequential processor to perform the compilation or optimization process. There are also issues of finding the right granularity to size subkernels, and the best candidate blocks to include in them. It is likely that these parameters will depend on the structure of the kernel.

9.3 Conclusion

This dissertation introduces the Harmony execution model and the Ocelot dynamic compiler as tools for reducing the complexity of designing applications for heterogeneous systems. Applications are written in a high level language using kernels similar to CUDA and OpenCL and then compiled into kernels, control decision, and variables primitives in the Harmony execution model. Harmony programs are executed by a runtime that creates a dynamic mapping between kernels and available processors. The runtime uses the Ocelot dynamic compiler to generate different binaries for heterogeneous processors. Preliminary results show that there is a significant amount of kernel level parallelism in Harmony applications that can be used to scale performance as processors are added to a system. Additional results show that the same Bulk-Synchronous-Parallel representation of a kernel can be efficiently compiled to CPUs and GPUs.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The next generation of intel ixp network processors. 2002.

[3] Susanne Albers. Better bounds for online scheduling. In *SIAM JOURNAL ON COMPUTING*, pages 130–139, 1997.

[4] AMD. Brook. `http://developer.amd.com/gpuassets/AMD-Brookplus.pdf`, 2007.

[5] AMD. R600/r700/evergreen assembly language format. Technical report, 2009.

[6] AMD. Opencl: The open standard for parallel programming of gpus and multi-core cpus. `http://ati.amd.com/technology/streamcomputing/opencl.html`, 2010.

[7] APPLE. ipad technical specifications and accessories for ipad. http://www.apple.com/ipad/specs/, 2009.

[8] Nitin Arora, Aashay Shringarpure, and Richard W. Vuduc. Direct n-body kernels for multicore platforms. In *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, pages 379–387, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. *Lecture Notes in Computer Science*, 3109:400–408, 2004.

[10] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, pages 163–174, 2009.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[12] B. Banerjia, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Dolby, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.

[13] M. Bentsen and B. Vinter. Enjing - a jit backend for cuda devices. 2010.

[14] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM.

[15] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46:720–748, September 1999.

[17] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task

farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM.

[18] Dan Bornstein. Dalvik virtual machine internals. Google I/O 2008, Juni 2008.

[19] Doug Burger and James R. Goodman. Billion-transistor architectures. *Computer*, 30(9):46–49, 1997.

[20] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[21] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM.

[22] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *In 26th Annual International Symposium on Computer Architecture*, 1999.

[23] William W. Carlson, Jesse M. Draper, and David E. Culler. *Introduction to UPC and Language Specification*, 1999.

[24] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[25] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.

[26] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[27] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing, 2009.

[28] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal on Research and Development*, 51:559–572, September 2007.

[29] Anton Chernoff and Ray Hookway. Digital fx!32 running 32-bit 86 applications on alpha nt. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 2–2, Berkeley, CA, USA, 1997. USENIX Association.

[30] Romain Cledat and Santosh Pande. Energy efficiency via the n-way model. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2011.

[31] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26:64–69, January 1983.

[32] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large windows/nt applications. In *Proc. MICRO29*, pages 80–89, 1996.

[33] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Barra: A parallel functional simulator for gpgpu. In *MASCOTS'10*, pages 351–360, 2010.

[34] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in gpgpu computations. In *Proceedings of the 2009 international conference on Parallel processing*, Euro-Par'09, pages 46–55, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, September 2008.

[36] ARM Coporation. The arm cortex-a9 processors. Technical report, 2009.

[37] NVIDIA Corp. Ptx: Parallel thread execution. `http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`, 2009.

[38] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[39] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[40] R.E. Diaconescu and H.P. Zima. An approach to data distributions in chapel. 21:313–335, 2007.

[41] Gregory Diamos. The design and implementation ocelot's dynamic binary translator from ptx to multi-core x86. Technical report, CERCS, 2009.

[42] Gregory Diamos. State explosion: An obvious limitation to strong scaling. Technical report, NFinTes, 2009.

[43] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Haicheng Wu, Andrew Kerr, and Sudhakar Yalamanchili. Simd re-convergence at thread frontiers. Technical report, 2011.

[44] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating gpu binaries to tiered simd architectures with ocelot. Technical report, 2009.

[45] Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. Ocelot: A dynamic compilation framework for ptx. `http://code.google.com/p/gpuocelot/`, 2009.

[46] Gregory Diamos and Sudakhar Yalamanchili. Speculative execution on Multi-GPU systems. In *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[47] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *HPDC'08*, Boston, Massachusetts, USA, June 2008. ACM.

[48] Rodrigo Dominguez, David R. Kaeli, John Cavazos, and Mike Murphy. Improving the open64 backend for gpus. Technical report, Dept. of Electrical and Computer Engineering, 2009.

[49] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[50] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[51] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[52] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop Scheduling. *Mathematical Operations Research*, 1:117–129, 1976.

[53] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to intel core duo processor architecture. *Intelligence/sigart Bulletin*, 2006.

[54] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–, Washington, DC, USA, 1998. IEEE Computer Society.

[55] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support*

for programming languages and operating systems, pages 151–162, New York, NY, USA, 2006. ACM.

[56] R. L. Graham and R. L. Grahamt. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[57] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2009.

[58] The Portland Group. Cuda fortran, 2010.

[59] The Portland Group. *PGI Fortran and C Accelerator Programming Model*, 2010.

[60] Vinod Grover, Sean Lee, and Andrew Kerr. Plang: Translating nvidia ptx language to llvm ir machine, 2009.

[61] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[62] Tsuyoshi Hamada and Naohito Nakasato. Infiniband trade association, infiniband architecture specification, volume 1. In *International Conference on Field Programmable Logic and Applications*, pages 366–373.

[63] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[64] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[65] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[66] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[67] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[68] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[69] Amir Hormati, Yoonseo Choi, Mark Woh, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Macross: Macro-simdization of streaming application. In *ASPLOS-XV: Proceedings of the 15th international conference on Architectural support for programming languages and operating systems*, Pittsburg, PA, USA, 2010.

[70] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan1, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege,

J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC10: IEEE International Solid-State Circuits Conference*, 2010.

[71] Buck Ian, Foley Tim, Horn Daniel, Sugerman Jeremy, Fatahalian Kayvon, Houston Mike, and Hanrahan Pat. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[72] IMPACT. The parboil benchmark suite, 2007.

[73] Intel. Ct: C for throughput computing.

[74] Intel. Intel atom processor n400 series. http://download.intel.com/design/processor/datashts/322847.pdf, 2009.

[75] Intel. *Intel HD Graphics OpenSource Programmer Reference Manual*, June 2010.

[76] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 248–256, New York, NY, USA, 2010. ACM.

[77] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[78] Mike Johnson. *Superscalar microprocessor design*. Prentice Hall series in innovative technology. Prentice Hall, 1991.

[79] I T Jolliffe. *Principal Component Analysis*. Springer, 2002.

[80] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, and S. Naffziger. An x86-64 core implemented in 32nm soi cmos. In *ISSCC10: IEEE International Solid-State Circuits Conference*, 2010.

[81] Henry Kaiser. The varimax criterion for analytic rotation in factor analysis. *Psychometrika*, 23:187–200, 1958. 10.1007/BF02289233.

[82] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. Technical report, Champaign, IL, USA, 1993.

[83] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23:66–76, March 2003.

[84] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[85] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Submitted to Third Workshop on General-Purpose Computation on Graphics Procesing Units*, Pittsburg, PA, USA, March 2010.

[86] R. E. Kessler. The alpha 21264 microprocessor: Out-of-order execution at 600 mhz. 1998.

[87] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computing*, 22:786–793, August 1973.

[88] Christoforos E. Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanovic, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Katherine Yelick. Scalable processors in the billion-transistor era: Iram. *Computer*, 30(9):75–78, 1997.

[89] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2008. ACM.

[90] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, Washington, DC, USA, 2004. IEEE Computer Society.

[91] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Sigplan Notices*, pages 318–328, 1988.

[92] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[93] Sean Lee, Vinod Grover, Manuel M. T. Chakravarty, and Gabriele Keller. Gpu kernels as data-parallel array computations in haskell. Technical report, 2009.

[94] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[95] ChiKeung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO'09*, New York, USA, devember 2009. IEEE.

[96] Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the darpa hpcs language project. *Parallel Processing Letters*, 17(1):89–102, 2007.

[97] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6:200–209, April 1962.

[98] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A fine-grain thread decomposition scheme for speculative multithreading. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 15–25, Washington, DC, USA, 2009. IEEE Computer Society.

[99] S.A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65 –68, nov. 2007.

[100] Jos F. Martnez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *In Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, 2002.

[101] Scott Mcfarling. Combining branch predictors. Technical report, Digital Western Laboratory, June 1993.

[102] Wen mei W. Hwu and David Kirk. Proven algorithmic techniques for many-core processors. `http://impact.crhc.illinois.edu/gpucourses/courses/sslecture/lecture2-gather-scatter-2010.pdf`, 2011.

[103] Erik Meijer, Redmond Wa, and John Gough. Technical overview of the common language runtime, 2000.

[104] Duane Merrill and Andrew Grimshaw. Revisiting sorting for gpgpu stream architectures. Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[105] Giridhar Sreenivasa Murthy, Muthu Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for gpgpu programs. In *24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, Georgia, USA, 4 2010.

[106] Hashem H. Najaf-abadi and Eric Rotenberg. Architectural contesting: exposing and exploiting temperamental behavior. *SIGARCH Comput. Archit. News*, 35(3):28–35, 2007.

[107] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[108] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[109] Yuri Nishikawa, Michihiro Koibuchi, Masato Yoshimi, Akihiro Shitara, Kenichi Miura, and Hideharu Amano. *Performance Analysis of ClearSpeeds CSX600 Interconnects*, pages 203–210. 2009.

[110] NVIDIA. Cuda reference manual. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf`, 2009.

[111] NVIDIA. Nvidias next generation cuda compute architecture: Fermi. Technical report, 2009.

[112] NVIDIA. Nvidia opencl jumpstart guide. `http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf`, 2010.

[113] NVIDIA. User guide: Tegratm 200 series. `http://developer.download.nvidia.com/tegra/docs/Tegra%20200_Series_DevBoard_UserGuide_DG04927001v01.pdf`, 2010.

[114] Kunle Olukotun, Jules Bergmann, Kun Chang, and Basem A. Nayfeh. Rationale, design and performance of the hydra multiprocessor. Technical report, Stanford, CA, USA, 1994.

[115] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[116] Pablo Montesinos Ortego and Paul Sack. Sesc: Superescalar simulator. Technical report, 2004.

[117] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[118] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Computer Architecture News*, 25:206–218, May 1997.

[119] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. High-performance cuda kernel execution on fpgas. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 515–516, New York, NY, USA, 2009. ACM.

[120] David B. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, 16:8–15, April 1996.

[121] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29:66:1–66:13, July 2010.

[122] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, and Jared Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, 1997.

[123] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 412–423, New York, NY, USA, 2007. ACM.

[124] Gallagher Pryor, Brett Lucey, Sandeep Maddipatla, Chris McClanahan, John Melonakos, Vishwanath Venugopalakrishnan, Krunal Patel, Pavan Yalamanchili, and James Malcolm. High-level gpu computing with jacket for matlab and c/c++. Technical report, 2010.

[125] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20(4):47–57, 2000.

[126] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10:123–165, June 1978.

[127] Helge Rhodin. A PTX Code Generator for LLVM, October 2010.

[128] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[129] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus, gpus and intel mic architectures. Technical report, Intel Labs, 2010.

[130] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27, 2008.

[131] PCI SIG. *PCI Local Bus Specification, revision 2.3*, 2002.

[132] PCI SIG. *PCIe Base Specification 3.0*, 2010.

[133] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30(9):68–74, 1997.

[134] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[135] H. Spaanenburg. Multi-core/tile polymorphous computing systems. pages 1 –4, may 2008.

[136] G. B. Steven, S.M. Gray, and R. G. Adams. Harp: a parallel pipelined risc processor. *Microprocessors and Microsystems*, 13:579–587, November 1989.

[137] John Stratton, Vinod Grover, Jaydeep Marathe, Baastian Aarts, Mike Murphy, Ziang Hu, and Wen mei Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *CGO 2010*, Toronto, Canada, April 2010.

[138] John Stratton, Sam Stone, and Wen mei Hwu. Mcuda: An efficient implementation of cuda kernels on multi-cores. Technical Report IMPACT-08-01, University of Illinois at Urbana-Champaign, March 2008.

[139] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[140] Samantika Subramaniam, Anne Bracy, Hong Wang 0003, and Gabriel H. Loh. Criticality-based optimizations for efficient load processing. In *HPCA*, pages 419–430, 2009.

[141] David Tarjan, Jiayuan Meng, and Kevin Skadron. Increasing memory miss tolerance for simd cores. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[142] K. Thangarajan, W. Mahmoud, E. Ososanya, and P. Balaji. Survey of branch prediction schemes for pipelined processors. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pages 324 – 328, 2002.

[143] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[144] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[145] TidePowered. Gpu.net: A system that allows developers to write their gpu code in any .net-supported language., 2011.

[146] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[147] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[148] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[149] Kenton Varda. Protocol buffers. `http://code.google.com/apis/protocolbuffers/`.

[150] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[151] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software: The raw machine. Technical report, Cambridge, MA, USA, 1997.

[152] C. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, 1964.

[153] Henry Wong, Anne Bracy, Ethan Schuchman, Tor M. Aamodt, Jamison D. Collins, Perry H. Wang, Gautham Chinya, Ankur Khandelwal Groen, Hong Jiang, and Hong Wang. Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 52–61, New York, NY, USA, 2008. ACM.

[154] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd Workshop on Languages and Compilers for Parallel Computing (LCPC)*, october 2009.

[155] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In Guang Gao, Lori Pollock, John Cavazos, and Xiaoming Li, editors, *Languages and Compilers for Parallel Computing*, volume 5898 of *Lecture Notes in Computer Science*, pages 172–187. Springer Berlin, 2010.

[156] D. C. Youla and H. Webb. Image restoration by the method of convex projections: Part 1: Theory. *Medical Imaging, IEEE Transactions on*, 1(2):81 –94, oct. 1982.

[157] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *in ISPASS 07*, 2007.

[158] Fubo Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.*, pages 231–245, 2004.

[159] Xin David Zhang. A streaming computation framework for the cell processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2007.